Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional

Unidad Zacatenco

Departamento de Computación

Diseño de un nuevo algoritmo basado en evolución diferencial para optimización global a gran escala y su implementación eficiente en GPUs

T E S I S

Que presenta

Oscar Pacheco Del Moral

Para obtener el Grado de

Maestro en Ciencias en Computación

Director de la Tesis:

Dr. Carlos Artemio Coello Coello

Ciudad de México **Octubre, 2020**

Resumen

Una gran variedad de problemas del mundo real pueden ser formulados de tal forma que sea necesario encontrar el óptimo global (o la mejor aproximación a él) de una función sujeta a varios parámetros o variables de decisión en un espacio de búsqueda continuo. Este tipo de problemas son conocidos como problemas de optimización global (OG) y varias técnicas han sido desarrolladas para su solución, siendo mayormente aplicadas las metaheurísticas debido las diversas restricciones de uso de los métodos exactos (por ejemplo, aquellos basados en el gradiente de la función a optimizarse). Dentro del mundo de las metaheurísticas existe una familia de algoritmos, llamados evolutivos (AEs), los cuales han sido aplicados a problemas de OG por décadas, destacándose el algoritmo de evolución diferencial (ED).

La ED es un algoritmo bio-inspirado, basado en población y de naturaleza estocástica cuyo primer paso es inicializar una población de forma aleatoria. Después, durante generaciones (empleando dos operadores especiales), guía a los individuos hacia un óptimo global. Sin embargo, existe evidencia experimental indicando que el rendimiento de la ED (y de las metaheurísticas en general) se deteriora drásticamente al aumentar la dimensionalidad del problema, particularmente cuando el problema cuenta con cientos o miles de variables. Este tipo de problemas son denominados de optimización global a gran escala (OGGE).

Se han propuesto varios algoritmos para la resolución de problemas a gran escala. Las propuestas que mejores resultados han mostrado generalmente siguen un enfoque de diseño basado en descomposición (divide y vencerás) o un enfoque de diseño sin descomposición pero basado en el concepto de hibridación el cual consiste en combinar las mejores características de dos o más algoritmos para formar uno nuevo buscando que supere a cualquiera de sus componentes. La ED ha sido empleada en ambos casos, destacándose su uso en algoritmos híbridos como componente de búsqueda global (exploración) el cual usualmente está acoplado a un componente de búsqueda local (explotación). En estos casos, la etapa de explotación es generalmente manejada por un método de búsqueda local puro (no basado en población) el cual puede o no estar basado en el uso del gradiente. Respecto a su uso en algoritmos basados en descomposición, la ED se suele emplear como optimizador para resolver los subproblemas.

En esta tesis proponemos un nuevo algoritmo basado en la evolución diferencial, par-

ticularmente en una de sus variantes adaptativas conocida como **evolución diferencial empleando adaptación de parámetros basado en el historial de éxito** (SHADE, por sus siglas en inglés). La nueva propuesta no sigue un enfoque de diseño basado en descomposición ya que consideramos relevante la tarea de escalar la ED a problemas de alta dimensionalidad sin recurrir al paradigma divide y vencerás. Así mismo, la idea subyacente al nuevo diseño no es explotar el potencial de combinar a la ED con otros métodos sino el potencial de combinar diferentes variantes de la ED. Como resultado, la nueva propuesta (denominada GL-SHADE) integra dos poblaciones que colaboran entre sí durante el proceso de optimización; la primera de ellas cambia acorde con un esquema de evolución especializado en búsqueda global mientras que la segunda población evoluciona de acuerdo con un esquema especializado en búsqueda local. La interacción entre poblaciones es llevada a cabo mediante un operador sencillo de migración. Adicionalmente, la implementación de GL-SHADE es llevada a cabo mediante el uso de GPUs con el fin de acelerar su tiempo de ejecución promedio.

Los resultados experimentales obtenidos (se emplea el conjunto de funciones de prueba CEC'13 LSGO) indican que la nueva propuesta es competitiva con respecto a algoritmos del estado del arte en el área de OGGE además de exhibir un desempeño superior al del mejor algoritmo no basado en descomposición e híbrido conocido hasta el momento (el cual hace uso de la ED como componente de búsqueda global).

Adicionalmente, los estudios estadísticos elaborados muestran que el algoritmo de GL-SHADE exhibe un mejor desempeño que cualquiera de sus bloques constructores indicando que la combinación adecuada de diferentes estrategias de evolución es capaz de potenciar el motor de búsqueda tanto como la combinación de la ED con diferentes algoritmos. Finalmente, las pruebas de tiempo efectuadas sobre las implementaciones en paralelo (`CUDA`) y secuencial (`C++`) de GL-SHADE indican que la aceleración alcanzada depende del problema de prueba adoptado como función objetivo. No obstante, en la mayoría de los casos la implementación paralela de GL-SHADE logra una aceleración de al menos 2x respecto a su implementación secuencial.

Abstract

A great variety of world problems can be formulated in such a way that it is necessary to find the global optimum (or the best approximation to it) of a function subject to several decision variables on a continuous search space. These are known as global optimization (GO) problems and several techniques have been developed to solve them. Metaheuristics have been mostly applied in them, given the many restrictions for using exact methods (e.g., those requiring the gradient of the function to be optimized). Within the realm of metaheuristics the so-called evolutionary algorithms (EAs) have been applied to GO problems for decades, from which differential evolution (DE) has been widely adopted.

DE is a stochastic, population-based, bio-inspired algorithm whose first step is to initialize a population randomly. Then, for generations (using two special operators), it guides individuals toward a global optimum. However, there is experimental evidence indicating that the performance of DE (and of metaheuristics in general) deteriorates drastically when increasing the dimensionality of the problem, particularly when the problem has hundreds or thousands of variables. These are known as large-scale global optimization (LSGO) problems.

Several algorithms have been proposed for solving large-scale problems. The proposals that have shown the best results generally follow a design based on decomposition (divide-and-conquer) or a design based on the concept of hybridization, which consists of combining the best characteristics of two or more algorithms to form a new one aiming to overcome any of its single components. DE has been adopted in both cases, highlighting its use in hybrid algorithms as a global search component (exploration) which is usually coupled to a local search component (exploitation). In these cases, the exploitation stage is generally managed by a pure (non-population based) local search method which may or may not be gradient-based. Regarding its use in decomposition-based algorithms, it has been adopted as an optimizer to solve subproblems.

In this thesis we propose a new algorithm based on differential evolution, particularly in one of its adaptive variants known as **differential evolution using the adaptation of parameters based on the history of success** (SHADE). The new proposal does not follow a decomposition-based design approach as we aimed to scale DE to high dimensionality problems without resorting on the divide-and-conquer paradigm. Likewise, the idea behind the new design is not to exploit the potential of combining DE with other

methods, but the potential to combine different DE variants. As a result, the new proposal (called GL-SHADE) integrates two populations that collaborate with each other during the optimization process. The first of them changes according to a scheme specialized in global search while the second population evolves according to a scheme specialized in local search. The interaction between populations is carried out using a simple migration operator. Furthermore, the implementation of GL-SHADE is carried out using GPUs in order to speed up its average execution time.

The experimental results obtained (the set of test functions CEC'13 LSGO was adopted) indicate that the new proposal is competitive with respect to several state-of-the-art LSGO algorithms. Additionally, the proposed approach has a better performance than that of the best algorithm known so far, which is not based on decomposition and is hybrid (it makes use of DE as a global search component).

Additionally, the statistical studies carried out show that the proposed GL-SHADE algorithm exhibits a better performance than any of its building blocks, indicating that the appropriate combination of different evolution strategies is capable of enhancing the search engine analogously to the combination of DE with different methods. Finally, the tests carried out on the GL-SHADE parallel (`CUDA`) and sequential (`C++`) implementations indicate that the speed up achieved depends on the test problem adopted. However, in most cases, a speed up of at least 2x was achieved when comparing the execution time of the sequential and parallel implementations of GL-SHADE.

Agradecimientos

En primer lugar, agradezco a Dios, por darme la salud que me permitió culminar una de las etapas más importantes de mi vida personal y profesional.

Agradezco infinitamente a mis padres, Martha Del Moral y Rodolfo Pacheco, por sus interminables consejos, sus valores, por sus esfuerzos constantes que me han permitido ser una persona de bien, pero más que nada, por su apoyo y amor incondicional sin los cuales posiblemente no sería la persona que soy. Es así que este trabajo es una dedicatoria a mis padres.

Agradezco enormemente al Dr. Carlos Coello, por sus constantes enseñanzas y guía sin las cuales este trabajo no sería posible. Su gran labor como investigador, su paciencia, accesibilidad, disposición, experiencia y compromiso con la excelencia me permitieron concluir esta maestría.

Agradezco a mis revisores de tesis, Dr. Luis Gerardo de la Fraga y Dr. Amilcar Meneses, por el tiempo invertido y sus comentarios que permitieron mejorar este trabajo.

Agradezco al CINVESTAV y a CONACyT por el apoyo económico otorgado a lo largo de mis estudios de posgrado. Este trabajo de tesis se derivó del proyecto CONACyT titulado "Esquemas de Selección Alternativos para Algoritmos Evolutivos Multi-Objetivo" (Ref. 1920 de la convocatoria *Fronteras de la Ciencia 2016*) cuyo responsable técnico es el Dr. Carlos A. Coello Coello.

Índice general

Índice de figuras

Índice de tablas

1 Introducción

En múltiples problemas del mundo real es común enfrentar tareas que pueden ser planteadas como un problema de optimización global de un solo objetivo, debido a que se requiere maximizar o minimizar una función que depende de múltiples variables de decisión. Algunas de las áreas en donde es común encontrar problemas de este tipo son: biología, ingeniería, minería de datos, aeronáutica, economía, ruteo de vehículos, entre muchas otras [1, 2, 3]. En los últimos años y debido a los avances tecnológicos, cada vez es más frecuente que surjan problemas de optimización con cientos e incluso miles de variables. A esta área se le conoce como optimización global a gran escala (OGGE). Dadas las restricciones de uso de la mayoría de los métodos exactos en problemas de OGGE y el pobre rendimiento de las metaheurísticas tradicionales en este tipo de problemas, se han planteado diversas propuestas para resolverlos [4]. La técnica de descomposición es una de las más empleadas, aunque también existen algoritmos que no están basados en ella. En este trabajo, presentamos una nueva propuesta que no emplea descomposición y que está basada únicamente en el algoritmo de evolución diferencial.

1.1. Antecedentes y motivación

Un problema de optimización global (OG) de un solo objetivo consiste en encontrar el óptimo global de una función que depende de múltiples variables de decisión. La solución del problema es normalmente un vector de variables aunque en algunos casos puede ser incluso un conjunto de ellos (por ejemplo, cuando el problema es multimodal) [5]. Existen dos clases de algoritmos para atacar un problema de OG: (1) exactos y (2) estocásticos. Los algoritmos exactos tienen la garantía de encontrar el óptimo global en una cantidad finita de tiempo. Sin embargo, son imprácticos en la mayoría de los casos, por lo que los algoritmos estocásticos se posicionan como una alternativa razonable a pesar de que generalmente no garanticen encontrar el óptimo global. Los algoritmos evolutivos (AEs) son un tipo especial de algoritmos estocásticos que han mostrado excelentes habilidades de búsqueda al aplicarlos a diversas clases de problemas de OG.

Dentro de los AEs existe uno denominado evolución diferencial (ED) que se distingue de entre sus familiares (el algoritmo genético y la estrategia evolutiva, por ejemplo) por su excelente rendimiento en problemas de OG de baja y mediana dimensionalidad (alrededor

de 30-100 variables). Esto le ha permitido consolidarse como un algoritmo de renombre en el área [1].

Además de los problemas de OG de baja y mediana dimensionalidad, existen los problemas a gran escala donde el número de variables de decisión es al menos de 100 [6]. Los problemas de optimización global a gran escala (OGGE) suponen un desafío ya que generalmente los algoritmos estocásticos de búsqueda, incluida la ED, sufren un deterioro en su rendimiento ya que mientras el número de variables crece linealmente, el espacio de búsqueda crece exponencialmente. En el caso particular de la ED, su desempeño comienza a deteriorarse, de forma más pronunciada, cuando el número de variables de decisión va más allá de 500 [1].

Actualmente, existen múltiples algoritmos especialmente diseñados para atacar problemas de OGGE y muchos de ellos hacen uso de la ED (ver [3, 7, 6, 8, 9]) habiendo varios (ver [10, 11]) que están totalmente basados en ella. De cualquier forma, todas las propuestas existentes siguen uno de los siguientes enfoques [4, 12]:

- **Enfoques basados en descomposición empleando el marco de coevolución cooperativa (CC).** La idea intuitiva es descomponer el problema de OGGE en subproblemas (divide y vencerás) más pequeños que son más fáciles de resolver. Es decir, optimizar todos los subproblemas para lograr el propósito de optimizar el problema original. La resolución por un enfoque CC tiene la ventaja de ser fácilmente escalable, además de que los subproblemas pueden ser resueltos empleando metaheurísticas tradicionales. Por otro lado, como principal desventaja se tiene que el rendimiento es muy sensible a la estrategia de descomposición empleada y generalmente es pobre en problemas no separables.

- **Enfoques no basados en descomposición.** La idea es optimizar el problema de OGGE como un todo. Este tipo de algoritmos son construidos a partir de rediseñar, modificar, combinar o añadir nuevos mecanismos a algoritmos existentes. Algunos ejemplos son:

  1. Nuevos mecanismos para ajuste automático de parámetros.
  2. Agregar estrategias de búsqueda local o global.
  3. Introducción de estrategias estructuradas de población y migración (cuando el método es basado en población).
  4. Diseño de nuevos operadores.
  5. Combinar algoritmos exactos y estocásticos.

Cada enfoque tiene sus propias ventajas y desventajas. No obstante, el enfoque de CC parece más adecuado además de eficiente dada su estructura intrínsecamente paralela. En realidad, el diseño de un algoritmo de CC no es una tarea sencilla ya que el desempeño del algoritmo en estos casos es muy sensible a la técnica de descomposición empleada

por lo que diseñar un buen algoritmo de CC se reduce a emplear e incluso proponer un algoritmo eficiente de descomposición [13, 12] .

En este trabajo se tiene un interés particular en mejorar el desempeño de la ED en problemas a gran escala sin recurrir al marco de CC. No obstante, se discutirán los algoritmos de CC mencionando sus ventajas, desventajas e incluso el funcionamiento de algunos de ellos con el fin de contrastarlos con aquellos no basados en descomposición.

Respecto a los algoritmos no basados en CC que hacen uso de la ED, las propuestas más competitivas son algoritmos híbridos los cuales se caracterizan por combinar la ED (componente explorativo) con métodos de búsqueda local (componente explotativo). También existen propuestas no híbridas, es decir, totalmente basadas en la ED pero su desempeño es generalmente inferior que el de los algoritmos híbridos.

El principal problema de las propuestas basadas totalmente en la ED (por ejemplo [10, 11]) es su incapacidad de poder establecer un equilibro entre la búsqueda global y la búsqueda local lo cual depende de varios factores tales como el tamaño de la población, el valor de los parámetros evolutivos y el esquema evolutivo adoptado. El interés en este trabajo está centrado en desarrollar una nueva propuesta no basada en descomposición que sea competitiva comparada con los mejores algoritmos del área de OGGE (basados y no basados en el marco de CC) cuyo motor de búsqueda esté totalmente basado en la ED aprovechando al máximo la versatilidad característica de este algoritmo evolutivo.

1.2. Propuesta de solución

El diseño de la nueva propuesta está basado en una idea que generalmente adoptan la propuestas híbridas, denominada **estrategia de búsqueda (global-local)** la cual consiste en dividir el proceso de optimización en rondas o iteraciones de tal forma que en cada ronda se lleva a cabo una etapa explorativa y otra explotativa. No obstante, en este caso, y a diferencia de las propuestas híbridas, la etapa explotativa es manejada por la ED (en realidad una variante de ella).

Las propuestas basadas en la ED regularmente adoptan al menos dos estrategias de mutación una útil para promover la exploración y otra para promover la explotación y, en algunos casos se utiliza, más de un tipo de recombinación para evolucionar a una población de individuos (candidatos a solución). En este caso, se emplean dos poblaciones cada una de ellas con una estrategia de mutación y un tipo de recombinación diferentes, lo cual permite establecer dos esquemas diferentes de evolución para cada población.

La población 1 evoluciona acorde con un esquema evolutivo especializado en búsqueda global empleando el algoritmo de evolución diferencial basado en el historial de éxito (SHADE, por sus siglas en inglés) mientras que la población 2 evoluciona acorde con un esquema especializado en búsqueda local acorde con una nueva variante de SHADE denominada $\text{eSHADE}_{\text{ls}}$ la cual fue diseñada específicamente para explotar el espacio de búsqueda. Respecto a la interacción entre poblaciones, se emplea un operador sencillo de

migración.

1.3. Estructura de la tesis

Incluyendo este capítulo, el presente trabajo está compuesto por siete capítulos y cuatro apéndices.

En el capítulo 2 se define formalmente el problema de optimización global mono-objetivo y se presentan los métodos más comunes que existen para su resolución. Adicionalmente, se introduce el concepto de **problema a gran escala** en el contexto de optimización global mono-objetivo y se presentan los enfoques propuestos hasta ahora para atacar problemas de este tipo.

En el capítulo 3 se introduce, presenta y describe una de las familias de algoritmos estocásticos más importante para optimización conocida como **algoritmos evolutivos**.

En el capítulo 4 se estudia el algoritmo de **evolución diferencial** con sumo detalle; se discute el esquema de evolución clásico, los diferentes esquemas evolutivos que existen además del clásico, los tipos de mecanismos que existen para adaptar los parámetros evolutivos de la ED y las propuestas basadas en la ED que han sido desarrolladas para resolver problemas de optimización global a gran escala mono-objetivo.

En el capítulo 5 se discute la **nueva propuesta** algorítmica basada en la evolución diferencial; se presenta el diseño propuesto y se describen a detalle los algoritmos SHADE y $\text{eSHADE}_{\text{ls}}$.

En el capítulo 6 se presenta la metodología seguida para **evaluar experimentalmente** el desempeño de la nueva propuesta.

En el capítulo 7 se presentan las **conclusiones** obtenidas y el **trabajo futuro** que puede derivarse de esta tesis.

En el apéndice A se discute la **implementación** de la nueva propuesta. En este caso se propone una implementación paralela mediante el uso una GPU para mejorar el tiempo de ejecución promedio. En el apéndice B se presentan y describen cada una de las funciones incluidas en el **conjunto de funciones de prueba** empleado durante los experimentos realizados. Por su parte, el apéndice C contiene tablas donde pueden ser consultados los **resultados numéricos** derivados de las pruebas experimentales. Finalmente, el apéndice D contiene los valores obtenidos al aplicar la **prueba de suma de rangos de Wilcoxon** para validar el desempeño de la nueva propuesta.

2 | Optimización global mono-objetivo

El área de optimización global (OG) es una rama de las matemáticas que tiene como objetivo encontrar el máximo o mínimo de una función sobre un dominio dado [14].

El presente capítulo tiene como objetivo abordar los conceptos básicos relacionados con el tema de optimización global mono-objetivo, así como los distintos enfoques o metodologías que existen para resolver problemas de dicha naturaleza. En la sección 2.1 se define formalmente el problema de optimización global mono-objetivo. En la sección 2.2 se define el criterio de optimalidad para dicha clase de problemas. La sección 2.3 presenta los métodos más comunes que existen para resolver problemas de optimización global. En la sección 2.4 se define el problema de optimización global mono-objetivo a gran escala. Finalmente, en la sección 2.5 se presentan los enfoques más comunes que existen para resolver problemas de optimización global a gran escala.

2.1. Problema de optimización global mono-objetivo

El objetivo en un problema de optimización global es encontrar los mejores elementos posibles x^* de un conjunto X de acuerdo a un conjunto de criterios $F = \{f_1, f_2, .., f_n\}$ donde cada f_i representa una función objetivo.

Definición 2.1.1. (Función objetivo) [14]. Una función objetivo $f : X \mapsto Y$ con $Y \subseteq \mathbb{R}$ es una función matemática que está sujeta a optimización.

El codominio Y de una función objetivo así como su rango deben ser un subconjunto de los números reales ($Y \subseteq \mathbb{R}$). El dominio X de f es conocido como el espacio del problema y puede ser representado por cualquier tipo de elementos tales como números y listas de valores, entre muchos otros tipos de elementos. En este sentido, el área de optimización global comprende todas las técnicas que pueden ser empleadas para encontrar los mejores elementos x^* en X respecto a los criterios $f \in F$ [14].

De acuerdo al número de funciones objetivo que se consideran como criterio podemos clasificar al problema de optimización como mono-objetivo o multi-objetivo. En el presente trabajo centramos nuestro interés en problemas de optimización de un solo objetivo cuyo espacio del problema es $\mathbb{R}^D$ o un subconjunto de él.

Definición 2.1.2. (Problema de optimización global mono-objetivo). Un problema de optimización global de un solo objetivo se define de la siguiente manera:

$$\begin{array}{rl} \text{Optimizar} & f(\vec{x}) \\ \text{Sujeto a} & lb_j \leq x_j \leq ub_j, \quad j = 1, 2, ..., D \\ & g_k(\vec{x}) \leq 0, \quad k = 1, 2, ..., m \\ & h_l(\vec{x}) = 0, \quad l = 1, 2, ..., n \end{array} \tag{2.1}$$

donde:

- $\vec{x} \in \Omega$: es el vector de variables de decisión.
- $\Omega \subset \mathbb{R}^D$: es el conjunto de vectores D-dimensionales que cumple las restricciones (límite, igualdad y desigualdad) conocida como región factible.
- $\vec{lb}, \vec{ub} \in \mathbb{R}^D$: son los vectores de cota inferior y superior que definen las restricciones de límite, repectivamente.
- $f : \Omega \mapsto \mathbb{R}$: es la función objetivo.
- $g_k, h_l : \mathbb{R}^D \mapsto \mathbb{R}$: representa las m restricciones de desigualdad y las n restricciones de igualdad respectivamente, donde $m, n \geq 0$.

Existen situaciones donde no hay restricciones de igualdad y desigualdad. En esos casos, la región factible Ω está definida solamente por las restricciones de límite.

2.2. El óptimo en un problema de optimización global mono-objetivo

El óptimo en un problema de optimización, con un solo criterio f, es el máximo o mínimo de f (dependiendo del problema). Sin pérdida de generalidad, supondremos que todas las funciones objetivo se minimizan. En caso de que una función f esté sujeta a maximización, podemos minimizar -f.

Definición 2.2.1. (Máximo local) [14]. Un máximo local $\hat{x}_l \in X$ de una función objetivo $f : X \mapsto Y$ es un elemento de entrada con $f(\hat{x}_l) \geq f(x)$ para todos los x que forman parte del vecindario de $\hat{x}_l$.

Si $X \subseteq \mathbb{R}^D$, entonces se puede escribir:

$$\forall\, \hat{x}_l \; \exists\, \epsilon > 0 : \; f(\hat{x}_l) \geq f(x) \; \forall x \in X, \; |x - \hat{x}_l| < \epsilon. \tag{2.2}$$

Definición 2.2.2. (Mínimo local) [14]. Un mínimo local $\check{x}_l \in X$ de una función objetivo $f : X \mapsto Y$ es un elemento de entrada con $f(\check{x}_l) \leq f(x)$ para todos los x que forman parte del vecindario de $\check{x}_l$.

Si $X \subseteq \mathbb{R}^D$, entonces se puede escribir:

$$\forall\, \check{x}_l \; \exists\, \epsilon > 0 : \; f(\check{x}_l) \leq f(x) \; \forall x \in X, \, |x - \check{x}_l| < \epsilon. \tag{2.3}$$

Definición 2.2.3. (Óptimo local) [14]. Un óptimo local $x_l^\star \in X$ de una función objetivo $f : X \mapsto Y$ es un máximo o mínimo local.

Definición 2.2.4. (Máximo global) [14]. Un máximo global $\hat{x} \in X$ de una función objetivo $f : X \mapsto Y$ es un elemento de entrada con $f(\hat{x}) \geq f(x) \; \forall x \in X$.

Definición 2.2.5. (Mínimo global) [14]. Un mínimo global $\check{x} \in X$ de una función objetivo $f : X \mapsto Y$ es un elemento de entrada con $f(\check{x}) \leq f(x) \; \forall x \in X$.

Definición 2.2.6. (Óptimo global) [14]. Un óptimo global $x^\star \in X$ de una función objetivo $f : X \mapsto Y$ es un máximo o mínimo global.

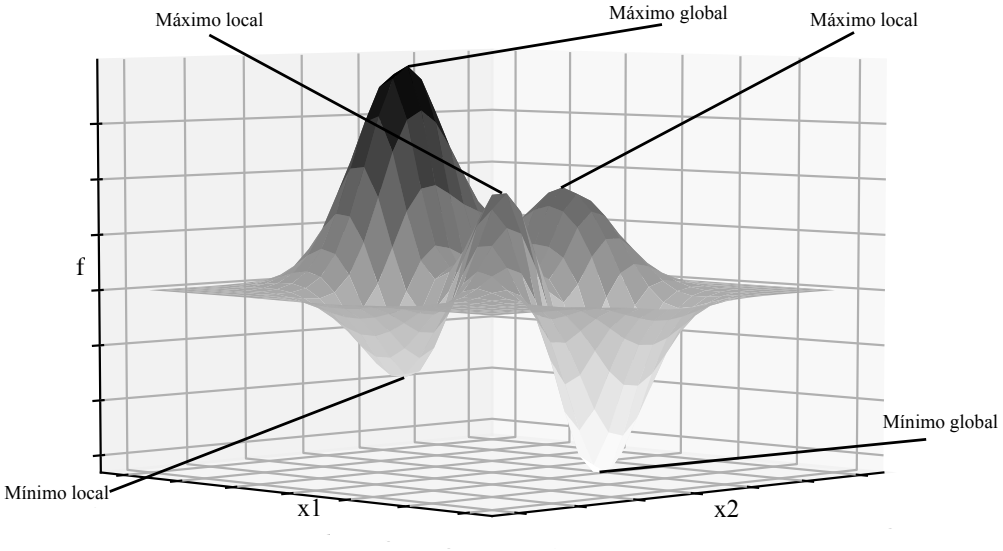

Figura 2.1: Óptimo global y local de una función bidimensional.

En la figura 2.1 se ilustra una función f definida sobre un espacio bidimensional ($X \subset \mathbb{R}^2$). Como se resalta en la gráfica, se distingue entre el óptimo global y los óptimos locales. Un óptimo global es un óptimo de todo el dominio X mientras que un

óptimo local es un óptimo de un subconjunto de X [14]. Para problemas de optimización global, claramente nuestro interés es encontrar el óptimo global.

La solución al problema de optimización suele ser un único vector de variables $\vec{x}$. Sin embargo, también es posible obtener un conjunto de soluciones (por ejemplo, en un problema multimodal) las cuales presentan el mismo valor al ser evaluadas con la función objetivo f.

Definición 2.2.7. (Conjunto óptimo) [14]. El conjunto óptimo es el conjunto que contiene todos los elementos óptimos o soluciones del problema.

Es así que la tarea de los algoritmos para optimización global es encontrar la mejor aproximación al conjunto óptimo o al menos un subconjunto de él [14].

2.3. Métodos de solución para problemas de optimización global mono-objetivo

Generalmente los algoritmos para optimización global pueden ser divididos en dos clases [14]:

1. Deterministas.
2. Estocásticos.

Los algoritmos deterministas, en teoría, tienen la garantía de encontrar al menos una de la soluciones óptimas, o todas ellas (en caso de problemas multimodales). Sin embargo, la carga computacional asociada a su uso se puede volver excesiva para problemas con un elevado número de variables o para funciones objetivo costosas (computacionalmente hablando) [15].

Se sabe que la mayoría de los modelos de optimización global y combinatoria de relevancia son NP-dificiles. Por lo tanto, incluso el aumento aparentemente "ilimitado" de poder computacional no resolverá su intratabilidad. Por esta razón, ante la presencia de alta dimensionalidad y sin conocimiento previo de la estructura del problema, se hace uso de algoritmos estocásticos exactos o puramente estocásticos [15].

Una familia especialmente relevante de los algoritmos puramente estocásticos son los enfoques basados en Monte Carlo, los cuales cambian la exactitud garantizada de la solución por un tiempo de ejecución más corto. Esto no significa que los resultados obtenidos al usarlos sean incorrectos; simplemente pueden no ser los óptimos globales [14]. Muchos de los métodos en este rubro son heurísticas y metaheurísticas inteligentes.

Definición 2.3.1. (Heurística) [16]. Técnica, método o procedimiento inteligente para realizar una tarea que no es producto de un riguroso análisis formal, sino de conocimiento experto sobre la tarea.

Definición 2.3.2. (Metaheurística) [16]. Las metaheurística son estrategias inteligentes para diseñar o mejorar procedimientos heurísticos muy generales con un alto rendimiento.

En la figura 2.2 (extraída de [14]) se esboza una taxonomía aproximada de los métodos de optimización global.

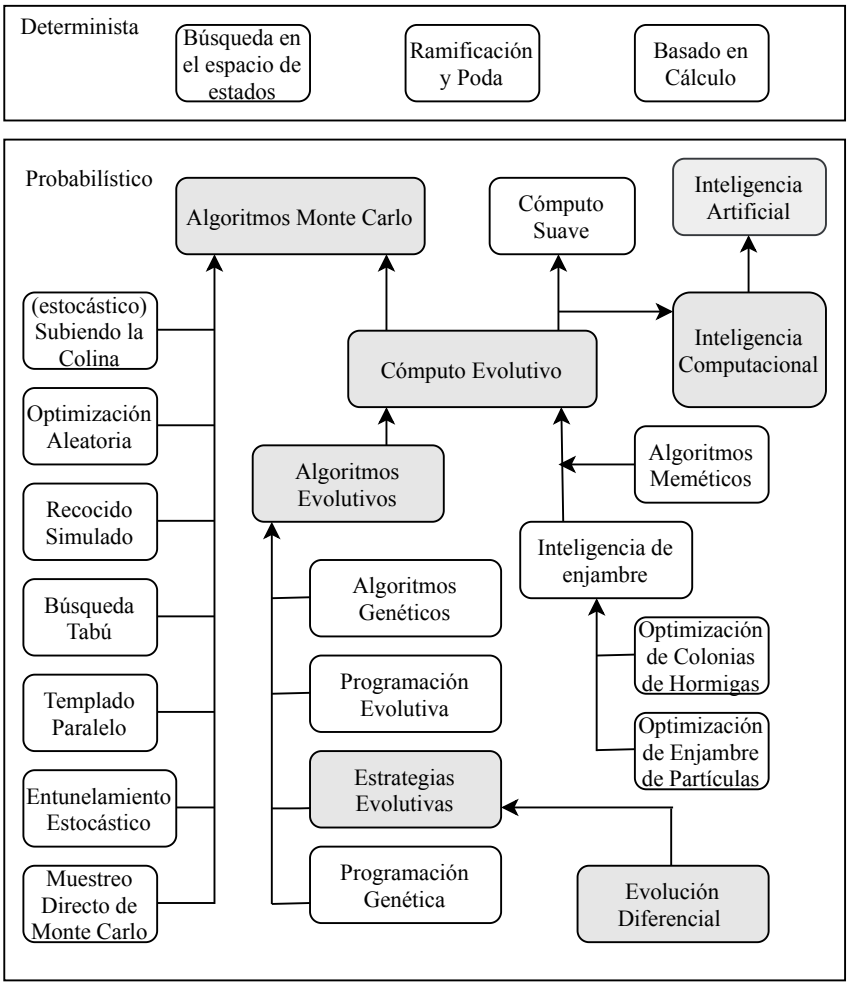

Figura 2.2: Taxonomía aproximada de los algoritmos de optimización global.

2.3.1. Métodos de solución deterministas

Son estrategias que guían la búsqueda de soluciones óptimas utilizando información del problema:

- **Algoritmos que hacen búsqueda en el espacio de estados:**

  - **Algoritmos glotones [17, 5]:** funcionan eligiendo la mejor solución disponible en cada iteración. Su mecanismo se basa en la suposición de que una solución óptima local formará parte de la solución óptima global. Sin embargo, cuando esto no ocurre, el algoritmo falla, devolviendo como solución un óptimo local.
  - **Algoritmos de búsqueda en profundidad [17]:** estos algoritmos eligen una solución y expanden posibles soluciones a partir de ella. Posteriormente, eligen alguna de las nuevas soluciones y repiten el procedimiento. Esto se lleva a cabo de forma iterativa hasta que no pueden seguir expandiendo una solución. En ese momento se reanuda la expansión de soluciones eligiendo al hermano del último nodo procesado. Esto se repite de forma recursiva hasta explorar todo el árbol.
  - **Algoritmos de búsqueda en amplitud [17]:** estos algoritmos son similares a los de búsqueda en profundidad en el sentido de ir eligiendo una asociación, generar nuevas soluciones y realizar la exploración del árbol producido de forma recursiva. Sin embargo, la principal diferencia es el orden de exploración del árbol. Los algoritmos de búsqueda en amplitud exploran el árbol una capa a la vez, es decir, todos los nodos con la misma profundidad. Tanto los algoritmos de búsqueda en profundidad como en amplitud se pueden considerar como algoritmos no informados, ya que realizan la búsqueda siempre en el mismo orden, sin importar la ubicación de la solución en el árbol.

- **Algoritmos de ramificación y poda [17]:** estos algoritmos buscan acotar el espacio de búsqueda para lo cual requieren el uso de heurísticas o algoritmos de decisión que proporcionen información del problema. El funcionamiento básico del algoritmo es buscar 'ramificar' soluciones que parecen prometedoras y 'podar' aquellas que no lo son. Esto significa que se generan posibles soluciones a partir de las mejores soluciones actuales y cuando alguna solución deja de ser prometedora se decide no expandir soluciones a partir de ella.

- **Algoritmos basados en cálculo [17]:** estos algoritmos requieren que el dominio de las variables sea continuo para poder aplicar técnicas de cálculo y así encontrar un valor óptimo.

Los métodos deterministas, en general, sufren de muchas restricciones de aplicabilidad en problemas del mundo real, por lo que usualmente no son viables [17]. Por tal motivo, se han desarrollado métodos estocásticos.

2.3.2. Métodos de solución estocásticos

Son estrategias de búsqueda con elementos estocásticos los cuales generalmente no hacen suposiciones sobre la estructura del problema:

- **Búsqueda aleatoria [5]:** es la estrategia de búsqueda estocástica más sencilla ya que simplemente evalúa un número dado de posibles soluciones de forma completamente aleatoria.

- **Recocido simulado [17]:** es una metaheurística inspirada en un fenómeno físico. En concreto, modela el proceso de enfriamiento de materiales conocido como recocido. Este proceso consiste en calentar un material (metal o líquido generalmente) hasta una temperatura determinada, después se mantiene la temperatura por un tiempo dado, y finalmente se deja enfriar el material gradualmente. El funcionamiento del recocido simulado es como sigue: Dada una solución actual, se elige de forma aleatoria otra solución que pueda ser generada desde la solución actual. Si la nueva solución mejora a la anterior, el algoritmo se mueve a la nueva solución. En caso contrario, el movimiento se realiza con alguna probabilidad $p < 1$. Dicha probabilidad se decrementa exponencialmente con el tiempo.

- **Búsqueda tabú [14, 17]:** la palabra *tabú* proviene de la Polinesia y describe un lugar u objeto sagrado. Las cosas que son tabú se deben dejar solas y no se pueden visitar ni tocar. La búsqueda tabú declara a los candidatos a solución que ya han sido visitados como tabú. Por lo tanto, no se deben volver a visitar pues así es menos probable que el proceso de optimización se atasque en un óptimo local. La puesta en marcha más simple de este enfoque es utilizar una lista tabú que almacene todos los candidatos a solución que ya han sido probados. Si se puede encontrar un candidato recién creado en esta lista, no se investiga y se rechaza de inmediato. La búsqueda tabú se caracteriza por evitar el principal problema de los métodos deterministas: encontrar únicamente óptimos locales.

- **Cómputo evolutivo [5]:** es un término genérico para varios métodos de búsqueda estocástica que simulan computacionalmente la evolución natural. Incorpora a los algoritmos genéticos, estrategias evolutivas, la programación genética y la programación evolutiva, conocidas colectivamente como algoritmos evolutivos (ver figura 2.2). Estas técnicas se basan en el principio Darwiniano de la **supervivencia del más apto**.

Los algoritmos evolutivos (AEs) se perfilan como una clase importante de métodos en el área de optimización global. Los AEs y, en general, los métodos de naturaleza estocástica no garantizan encontrar el óptimo global. Sin embargo, suelen producir buenas aproximaciones de él; es preferible tener a la mano una solución subóptima generada en un lapso de tiempo razonable a una óptima que requiere de varios años para ser encontrada.

2.4. Problema de optimización global a gran escala mono-objetivo

En los últimos años, y debido a los avances tecnológicos, cada vez es más común encontrar problemas del mundo real que pueden formularse como problemas de optimización global que involucran a cientos e incluso miles de variables. A este tipo especial de problemas se les denominan problemas de optimización global a gran escala (OGGE).

Definición 2.4.1. (Problema de optimización global a gran escala mono-objetivo). Es un problema de optimización global mono-objetivo donde el número de variables de decisión es mayor o igual a 1000 [6].

El aumento en la dimensionalidad hace que el proceso de optimización sea bastante más difícil. En consecuencia, los métodos tradicionales de solución dejan de ser efectivos [4]. Las principales razones son:

1. **Crecimiento del espacio de búsqueda:** mientras la dimensionalidad del problema crece linealmente, el espacio de búsqueda lo hace exponencialmente. Algunos autores se refieren a esto como "la maldición de la dimensionalidad" [4, 7].

2. **Cambio de las propiedades de la búsqueda:** una función unimodal a pequeña escala puede cambiar a una función multimodal cuando aumenta el número de dimensiones [7, 18].

3. **Aumento de la complejidad computacional:** evaluar una posible solución (empleando la función objetivo) suele ser costoso, lo que puede afectar el proceso de optimización, sin mencionar el incremento en complejidad intrínseco al algoritmo de optimización utilizado [7].

Así, el área de OGGE centra gran parte de su investigación en el desarrollo de nuevas técnicas o métodos que puedan ser empleados en problemas de alta dimensionalidad.

2.5. Enfoques de solución para problemas de optimización global a gran escala mono-objetivo

Cuando el número de variables de decisión del problema es muy elevado, los métodos deterministas, en general, no son considerados como una forma viable de atacar el problema [7]. Por tal motivo, los métodos estocásticos se consideran la forma más razonable de solución.

A pesar de los buenos resultados que han mostrado los métodos estocásticos (especialmente los AEs) en problemas de optimización a baja y mediana escala, su efectividad no

parece sostenerse a gran escala. Como consecuencia, se han propuesto nuevos enfoques de solución.

Los algoritmos evolutivos son una clase particularmente popular en optimización global, por lo que no sorprende que la mayoría de los algoritmos para OGGE estén basados en ellos. En la figura 2.3 (extraída de [4]) se muestran las principales estrategias de solución que adoptan los algoritmos basados en AEs para solucionar problemas de OGGE. Los dos posibles enfoques que se pueden seguir son: (1) el coevolutivo cooperativo (CC), es decir, el basado en descomposición o (2) el no basado en descomposición, es decir, el que soluciona el problema como un todo.

Si bien existen algoritmos para problemas a gran escala que no están basados en AEs (por ejemplo el método en [19]), prácticamente todas las propuestas destinadas al área de OGGE siguen uno de los dos enfoques ya mencionados.

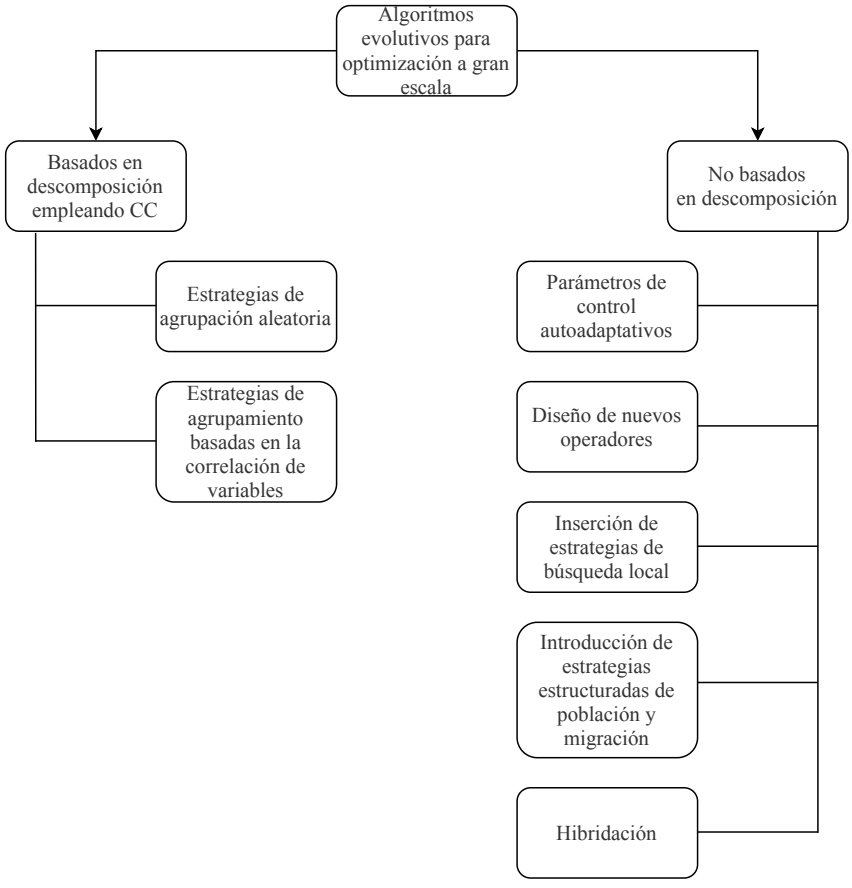

Figura 2.3: Enfoques empleados para atacar problemas a gran escala usando algoritmos evolutivos.

2.5.1. Enfoque basado en descomposición

En el enfoque basado en descomposición, la idea intuitiva es fragmentar todo un problema de optimización a gran escala en varios subproblemas más pequeños que son más fáciles de resolver para luego optimizarlos por separado y lograr así el propósito de optimizar el problema a gran escala; idea conocida como "divide y vencerás".

El método de coevolución cooperativa propuesto por Potter y De Jong (ver [20]) es un enfoque famoso (en el área del cómputo evolutivo) y común para descomponer problemas de optimización a gran escala [4].

Coevolución Cooperativa

El marco CC original sigue los siguientes pasos [21]:

1. Dividir los parámetros de la función objetivo (el problema original) en m subcomponentes de baja dimensión.

2. Optimizar cada uno de los m subcomponentes con un cierto AE. Tener en cuenta que el número de evaluaciones (de la función objetivo) está predeterminado.

3. Detener el proceso evolutivo una vez que se cumplan los criterios de detención o se haya excedido el número máximo de evaluaciones.

Los subcomponentes, a los que se denominan especies (ver figura 2.4), son implementados como subpoblaciones que siguen un proceso evolutivo de acuerdo al AE asignado. Los individuos de una especie en particular se pueden reproducir entre ellos pero no con individuos de otra especie (así como un caballo no se puede reproducir con un elefante).

La evaluación de cada individuo en una subpoblación se realiza concatenándolo con los mejores individuos del resto de las subpoblaciones manteniendo el orden original. Es decir, un individuo de la especie k corresponde al subcomponente k de izquierda a derecha (ver figura 2.4). El vector que resulta de la concatenación de los subcomponentes es conocido como el vector de contexto. El vector de contexto se alimenta a la función objetivo para la evaluación de la aptitud (aquí es donde ocurre la cooperación) [21]. Para que el enfoque CC sea efectivo, el tamaño de cada uno de los subcomponentes debe estar dentro de las capacidades de optimización del AE utilizado [21].

Por otro lado, una gran dificultad en la aplicación del método CC es la elección de una buena estrategia de descomposición. Se sabe que el rendimiento del método es potencialmente sensible al algoritmo de descomposición elegido [13, 4, 22]; generalmente basado en la **técnica de agrupamiento aleatorio** la cual trata de aumentar la probabilidad de colocar dos o más variables interactivas en una misma subpoblación. El problema de esta técnica es que no puede capturar la verdadera interacción entre parámetros. La razón es que el tamaño de un subcomponente es fijo y es el mismo para todos los subcomponentes

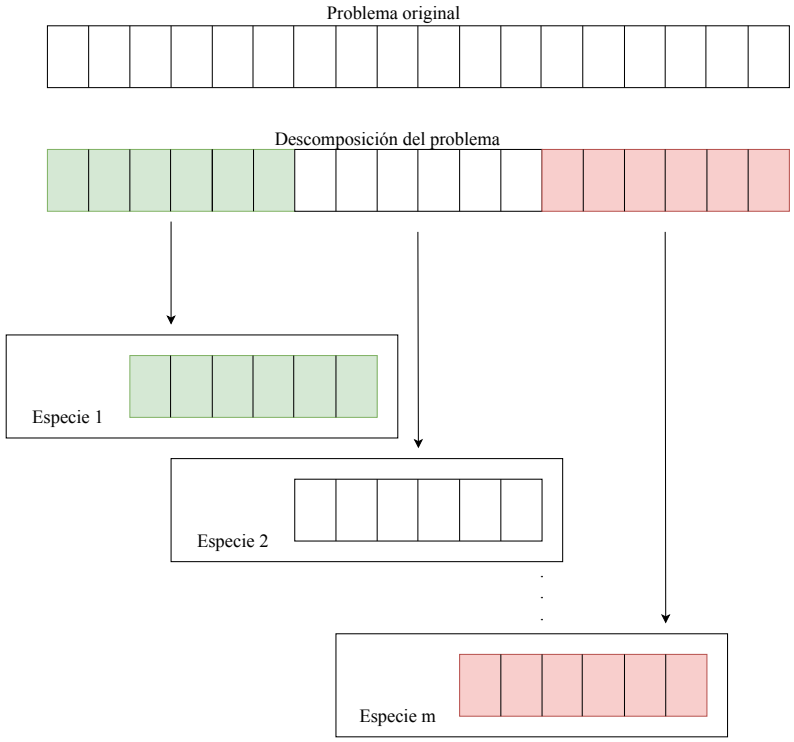

Figura 2.4: Descomposición usando coevolución cooperativa.

por lo que se podría argumentar que no todos los parámetros correlacionados forman tamaños de grupo fijos. Esto es cierto y el objetivo de la investigación en esta dirección es encontrar una mejor estrategia de agrupación para aumentar el rendimiento del enfoque CC [21].

En trabajos más recientes (a los lectores interesados en una lista actualizada de las técnicas de descomposición se les sugiere revisar [22]) se han desarrollado algoritmos de descomposición más efectivos basados en la correlación de variables (por ejemplo en [13, 23]). En dichos trabajos se propone una estrategia de descomposición automática llamada **agrupación diferencial** que puede descubrir la estructura de interacción subyacente de las variables de decisión y formar subcomponentes de modo que la interdependencia entre ellas se mantenga al mínimo. Actualmente, el desarrollo de nuevos algoritmos de descomposición es de interés en el área de OGGE.

2.5.2. Enfoque no basado en descomposición

En el enfoque no basado en descomposición, la idea intuitiva es resolver el problema como un todo. Existen diversas estrategias para mejorar el rendimiento de metaheurísticas tradicionales en problemas de alta dimensionalidad (por ejemplo en la figura 2.3 se muestran las estrategias que se consideran útiles para mejorar el rendimiento de los AEs

en problemas a gran escala sin usar descomposición), siendo la idea de hibridación una de las más populares [22].

Definición 2.5.1. (Hibridación). La hibridación, en el contexto de las metaheurísticas, se refiere principalmente al proceso de combinar las mejores características de dos o más algoritmos juntos, para formar un nuevo algoritmo que se espera que supere a los algoritmos originales en problemas de aplicación específicos o en problemas generales de referencia [1].

La forma en que diferentes metaheurísticas pueden ser combinadas depende del diseño del algoritmo, ya que existen diversos esquemas de combinación. Por ejemplo, en la figura 2.5 se puede observar un esquema de combinación secuencial (propuesto en [3]) que consiste en utilizar los algoritmos en secuencia, uno tras otro, cada uno de ellos reutilizando la salida del algoritmo anterior. Por lo tanto, el proceso de búsqueda general se divide en diferentes pasos (bloques de un número fijo de evaluaciones de la función objetivo) y la participación de cada algoritmo para el paso $i + 1$ se ajusta de acuerdo con su rendimiento en el paso anterior i (por participación se entiende el número de evaluaciones que tiene disponible para su ejecución). De acuerdo a la definición 2.5.1, $m \geq 2$ (ver figura 2.5).

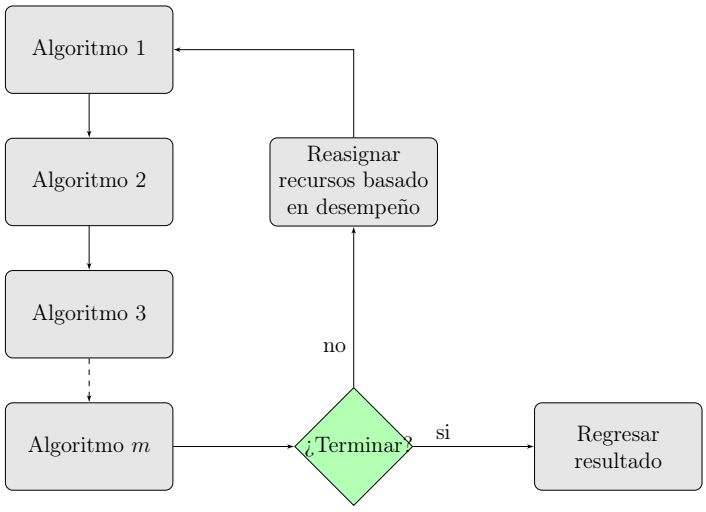

Figura 2.5: Esquema de combinación secuencial para un algoritmo híbrido.

Otra idea importante y relacionada con el concepto de hibridación es la inserción de estrategias de búsqueda global o local. Las metaheurísticas poblacionales (por ejemplo los AEs) pueden tener buena capacidad explorativa (búsqueda global) pero su capacidad explotativa (búsqueda local) es bastante limitada, mientras que las no basadas en población son buenas explotando pero no explorando. Al combinar dos o más metaheurísticas o algoritmos que se complementen (es decir, que presenten distintas capacidades de búsqueda) es posible mejorar el desempeño general [22].

Actualmente, el desarrollo de nuevos algoritmos híbridos es de interés para la comunidad relacionada con el área de OGGE. Uno de las varios temas de relevancia a este respecto es la hibridación de AEs con técnicas de programación matemática.

3 | Algoritmos Evolutivos

La computación evolutiva (CE) es un subcampo de las áreas de inteligencia artificial, aprendizaje automático y sistemas inteligentes, centrada en utilizar los principios de la evolución biológica para resolver problemas computacionales con diferentes complejidades [24]. Para la resolución de dichos problemas se hace uso de un conjunto especial de metaheurísticas bio-inspiradas conocidas en la literatura como algoritmos evolutivos (AEs). Su principal uso es en la solución de problemas de optimización global (aunque no exclusivamente), en donde el proceso mediante el cual se resuelve un problema puede verse análogo al proceso evolutivo [17].

En este contexto, el término evolución se refiere al proceso de optimización que trata de mejorar la capacidad de un organismo para vivir más tiempo en entornos cambiantes e inestables [24], los conceptos organismo y entorno son análogos a los de solución y problema, respectivamente.

Los AEs cuentan con una población inicial de organismos (también conocidos como individuos) que representan soluciones potenciales al problema. Durante la ejecución del algoritmo, dicha población evoluciona (cambia de forma iterativa) empleando mecanismos especiales bio-inspirados. La idea del proceso evolutivo es sustituir estocásticamente a los miembros menos aptos de la población por otros más aptos.

Si bien existen multiples métodos de optimización, la ventaja de los AEs es que hacen pocas suposiciones sobre la estructura del problema. Por lo tanto, funcionan consistentemente bien en diferentes categorías de problemas [14].

En este capítulo se presenta un panorama general de los principales AEs. En la sección 3.1 se proporciona una breve introducción donde se discuten los fundamentos biológicos que llevaron al desarrollo de metaheurísticas inspiradas en la evolución natural. También se presentan las principales metaheurísticas basadas en dichos principios y los conceptos básicos relacionados con ellos. En la sección 3.2 se presenta el algoritmo genético. En la sección 3.3 se discuten las estrategias evolutivas. En la sección 3.4 se aborda la programación evolutiva. Finalmente en la sección 3.5 se presenta la programación genética. La evolución diferencial se discute en el siguiente capítulo.

3.1. Introducción

Los orígenes de la computación evolutiva se remontan a principios de la década de 1950, cuando surgió la idea de utilizar los principios Darwinianos para la resolución automatizada de problemas [1].

3.1.1. Principios Darwinianos y fundamentos biológicos

La evolución Darwiniana es intrínsecamente un mecanismo robusto de búsqueda y optimización. Los problemas que las especies biológicas han resuelto se caracterizan por el caos, la probabilidad, temporalidad y una interacción no lineal. Estas también son características de problemas que han mostrado ser intratables por métodos clásicos de optimización (deterministas) [25]. La colección de teorías evolutivas más ampliamente aceptada es el paradigma neodarwinista. Dichas teorías afirman que la historia de la vida puede ser completamente representada por procesos físicos que operan sobre y dentro de poblaciones y especies. Estos procesos físicos son [25]:

- **Reproducción:** transferencia del material genético de un individuo (ya sea sexualmente o asexualmente) a su descendencia.
- **Mutación:** la existencia de errores de replicación durante la transferencia de material genético.
- **Competencia:** expansión de poblaciones en un espacio finito de recursos.
- **Selección:** liberar de la extinción a un grupo de individuos para que sigan compitiendo en el espacio de recursos.

Desde un punto de vista biológico, un individuo u organismo contiene material genético compuesto por ácido desoxirribonucleico (ADN). El material genético se emplea para guardar la información genética (**genotipo**) de una forma de vida orgánica y está almacenado en el núcleo de una célula. La información genética se organiza en estructuras llamadas **cromosomas** que son segmentos largos de ADN [17]. A su vez, un cromosoma se divide en múltiples **genes** los cuales son segmentos cortos de ADN y las unidades moleculares de la herencia. En al figura 3.1 se muestra una representación simplificada de un cromosoma y de un gen.

La información genética, la cual está conformada por genes, determina las características físicas y bioquímicas que tendrá un individuo. Se manifiestan a través de los rasgos físicos visibles (color de ojos, color de piel, funcionamiento del organismo, etc.). A este conjunto de características físicas se le denomina **fenotipo**. Se sabe que existen genes para manifestar las distintas características físicas. Sin embargo, lo que define o expresa dichos rasgos es la variación específica en la secuencia de ADN de un gen (**alelo**). Por

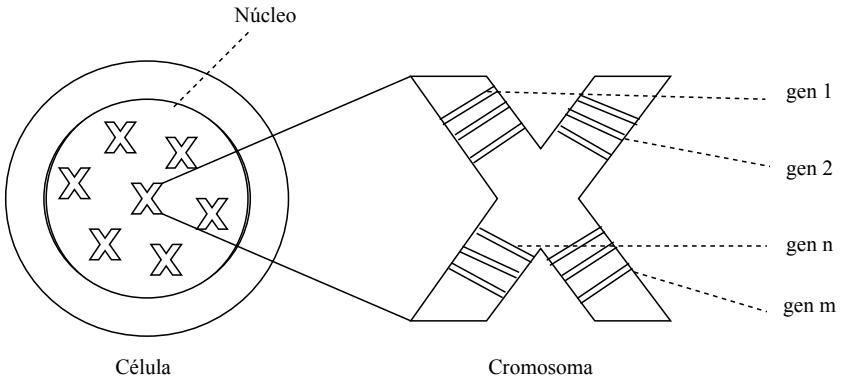

Figura 3.1: Modelo simplificado de cromosoma y gen.

ejemplo, existe un gen para el color de los ojos y el tipo de sangre por lo que existe un alelo para expresar ojos color azul/verde/café y tipos de sangre A/B/O.

Los algoritmos basados en el concepto evolutivo deben seguir, en mayor o menor medida, los siguientes fundamentos [25]:

1. El individuo es el principal objeto de la selección.
2. La variación genética es en gran medida un fenómeno aleatorio. Los procesos estocásticos juegan un rol muy importante en la evolución.
3. La variación genotípica es en gran medida producto de la reproducción o recombinación y solo en última instancia de la mutación.
4. La evolución "gradual" puede incorporar discontinuidades fenotípicas.
5. No todos los cambios fenotípicos son necesariamente consecuencias de la selección natural.
6. La evolución es un cambio en adaptación y diversidad, no precisamente un cambio en la frecuencia de los genes.
7. La selección es probabilísta, no determinista.

3.1.2. Orígenes de los algoritmos evolutivos

La idea de emplear los principios Darwinianos de la evolución natural surgió en los 1950s pero no fue si no hasta los 1960s que tres interpretaciones distintas de esta idea comenzaron a desarrollarse en tres lugares diferentes del mundo. La **programación evolutiva** (PE) fue introducida por Lawrence J. Fogel en Estados Unidos mientras que de forma simultánea Ingo Rechenberg y Hans-Paul Schwefel introdujeron las **estrategias**

evolutivas (EEs) en Alemania [1].

Por otro lado John Henry Holland, de la Universidad de Michigan en Ann Arbor, ideó un método independiente de simulación de la evolución Darwiniana para resolver problemas prácticos de optimización y lo llamó el **algoritmo genético** (AG) [1].

Estas áreas se desarrollaron por separado durante unos 15 años. Desde principios de la década de los 1990s se suele usar de manera genérica el término "algoritmo evolutivo" para referirse a este tipo de técnicas. A finales de los 1980s, surgió un cuarto paradigma dentro de la computación evolutiva que se considera una variante del algoritmo genético: la **programación genética** (PG) [1].

3.1.3. Conceptos básicos

Los algoritmos evolutivos emulan varios de los conceptos asociados a la teoría de la evolución natural, en concreto del paradigma neodarwinista. Los distintos algoritmos siguen, en mayor o menor medida, los fundamentos evolutivos asociados a la teoría, es decir, abstraen los principios básicos a diferentes niveles. No obstante, existen elementos comunes entre los diferentes algoritmos evolutivos [17]:

- **Individuo:** dado un problema específico, un individuo es la representación de una posible solución. Generalmente, se trata de una estructura de datos con múltiples parámetros que modelan una versión simplificada de un organismo biológico.
- **Población:** conjunto de individuos de la misma especie que pueden interactuar entre ellos mismos.
- **Generación:** es una iteración del algoritmo principal. En cada generación, se crean nuevos individuos mediante el uso de operadores evolutivos (los tres más populares son: mutación, cruza y selección).
- **Aptitud:** es un valor que cuantifica la calidad de un individuo como solución potencial al problema que se busca resolver. Permite determinar si un individuo es apto o no. En caso de serlo, tendrá una mayor probabilidad de sobrevivir o de tener descendencia. De lo contrario, tendrá una mayor probabilidad de ser eliminado de la población.
- **Operadores genéticos:**
  - **Mutación:** es un operador que forma un nuevo individuo mediante alteraciones, (usualmente pequeñas y llevadas a cabo de forma aleatoria) a la información de un individuo.
  - **Cruza:** es un operador que forma un nuevo individuo (llamado hijo) a partir de la información de dos o más individuos padres.

- **Selección:** es el proceso que determina cuáles individuos de la población pasan a la siguiente generación. Los individuos con mayor aptitud tienen más posibilidad de sobrevivir ya que la selección generalmente es llevada a cabo mediante un procedimiento estocástico.

3.2. Algoritmo Genético

El algoritmo genético fue propuesto por John H. Holland a principios de los 1960s [17]. El AG es un procedimiento que típicamente se implementa de la siguiente forma [25, 26]:

1. El problema que se quiere resolver debe ser representado por una función objetivo que indique la aptitud de cualquier solución candidata.

2. Se inicializa una población de soluciones candidatas (individuos) sujetas a ciertas restricciones (ver algoritmo 3.1, línea 1). Normalmente un individuo es codificado como un vector $\vec{x}$, el cual representa su genotipo, que está formado por una o más subcadenas las cuales codifican alguna característica en particular del problema. A dichas subcadenas se les llama cromosomas (ver figura 3.2). Cada elemento de un cromosoma se denomina gen y cada posible valor que puede llegar a tener un gen en una posición específica se denomina alelo. De acuerdo con la propuesta original de un AG, un individuo (solución) es tradicionalmente representado por una cadena binaria (ver figura 3.2), aunque también pueden emplearse números reales.
   En la figura 3.2 se observa un ejemplo de una codificación que representa una posible solución a un problema que depende de 4 parámetros donde cada uno puede ser codificado con 2 bits.

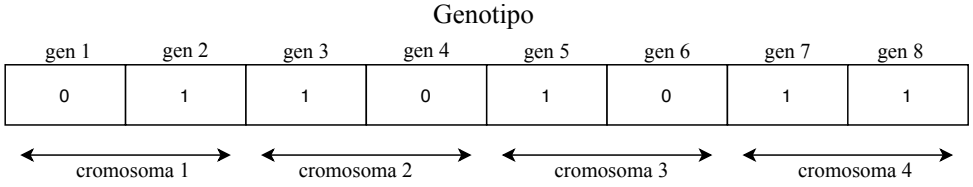

Figura 3.2: Ejemplo de representación binaria en un algoritmo genético.

3. Cada genotipo, $\vec{x}_i$, $i = 1, ..., P$, en la población es descodificado a una forma apropiada para ser evaluado con la función objetivo con el fin de que se le asigne un valor de aptitud (ver algoritmo 3.1, línea 2). El fenotipo de un individuo es la descodificación de su genotipo.

4. A cada genotipo, es decir, a cada individuo $\vec{x}_i$ le es asignado una probabilidad de selección para ser sometido a reproducción, p_i, $i = 1, ..., P$, de tal forma que la probabilidad de que un individuo sea seleccionado es proporcional a su aptitud respecto

a la de los otros individuos de la población.
Por ejemplo, si suponemos que la aptitud es un valor estrictamente positivo y deseamos maximizar dicho valor, es posible asignar la probabilidad de selección mediante la técnica de **selección por ruleta**:

$$p_i = \frac{\text{Aptitud}(\vec{x}_i)}{\sum_{k=1}^{P} \text{Aptitud}(\vec{x}_k)} \tag{3.1}$$

En la figura 3.3 se muestra un ejemplo de una posible configuración de la ruleta para una población compuesta de 4 individuos. De acuerdo a la ecuación (3.1), individuos con mayor aptitud tienen mayor probabilidad de ser seleccionados para ser sometidos a reproducción.
La selección por ruleta es la técnica más simple de selección proporcional, pero evidentemente existen varias formas adicionales de seleccionar individuos en un AG.

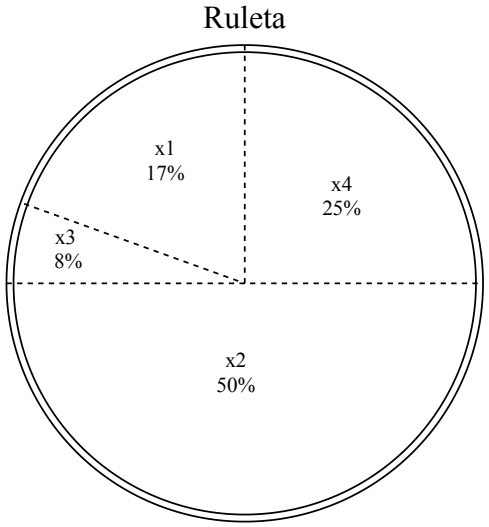

Figura 3.3: Posible configuración de ruleta para 4 individuos.

5. De acuerdo a las probabilidades asignadas de selección, p_i, $i = 1, ..., P$, una nueva población es generada (ver algoritmo 3.1, líneas 4-22) al seleccionar, de forma probabilística, individuos de la población actual (por ejemplo, aplicando la técnica de selección por ruleta). Los individuos seleccionados (llamados padres) generan nuevos individuos (llamados hijos) por medio de los operadores genéticos de cruza y mutación (ver algoritmo 3.1, líneas 6-17). El operador de cruza recibe como entrada dos padres y regresa como resultado dos hijos producto de la recombinación de las dos cadenas entrantes. El operador de cruza puede adoptar diferentes estrategias para llevar a cabo el procedimiento de recombinación. Las más populares (cruza de un punto, de dos puntos y uniforme) pueden ser consultadas en [26].

El operador de mutación es bastante más sencillo. Consiste en simplemente tomar a un individuo y por cada bit del cromosoma, se invoca una función que devuelve cierto con una determinada probabilidad (dada por el usuario). Si se obtiene cierto, el bit se cambia (si es 1 se vuelve 0 y si es 0 se vuelve 1); de lo contrario, el valor del bit permanece inalterado.

6. El proceso se detiene si el criterio de paro es satisfecho (ver algoritmo 3.1, línea 3). El criterio de paro puede definirse mediante un número máximo de generaciones, un número máximo de evaluaciones de la función objetivo o una falta de mejora de la mejor solución obtenida. Si el criterio de paro no se cumple, entonces el proceso va al paso 3 (donde se calcula la aptitud de los individuos de la población en curso) y el ciclo se repite.

A continuación se muestra el procedimiento que sigue un AG genérico con codificación binaria que supone un tamaño de población par:

Algoritmo 3.1: Algoritmo Genético

entrada: NP, pc, pm, A
salida : max A

1 Generar una población P de forma aleatoria compuesta por NP individuos
2 Para cada $\vec{x}_i \in P$, calcular $A(\vec{x}_i)$ // A es la función de aptitud
3 **mientras** no se cumpla el criterio de paro **hacer**
4 $k = 0$
5 **mientras** $k < (NP - 1)$ **hacer**
// Seleccionar con base en aptitud y siguiendo alguna estrategia de selección
6 $\text{padre}_1 = \text{seleccionar}(P)$
7 $\text{padre}_2 = \text{seleccionar}(P)$ // por ejemplo, selección por ruleta
// Cruzar a los individuos seleccionados siguiendo alguna estrategia de recombinación
8 **si** $\text{urnd}(\text{real}, 0, 1) \leq pc$ **entonces**
9 $\text{hijo}_1, \text{hijo}_2 = \text{cruzar}(\text{padre}_1, \text{padre}_2)$ // por ejemplo, cruza de un punto
10 **otro**
11 $\text{hijo}_1 = \text{padre}_1$ e $\text{hijo}_2 = \text{padre}_2$
// Mutar a los nuevos individuos generados
12 **para** $j = 0$ hasta $j <$ LargoCadena **hacer**
13 **si** $\text{urnd}(\text{real}, 0, 1) \leq pm$ **entonces**
14 mutar el j-ésimo gen de hijo_1
15 **para** $j = 0$ hasta $j <$ LargoCadena **hacer**
16 **si** $\text{urnd}(\text{real}, 0, 1) \leq pm$ **entonces**
17 mutar el j-ésimo gen de hijo_2
// Añadir nuevos candidatos a la siguiente generación
18 $P^k_{\text{nuevo}} = \text{hijo}_1$, $P^{k+1}_{\text{nuevo}} = \text{hijo}_2$
19 $k = k + 2$
20 Para cada $\vec{x}_i \in P_{\text{nuevo}}$, calcular $A(\vec{x}_i)$
// Aplicar elitismo
21 Si el individuo más apto $\vec{\hat{x}}$ de P es mejor que el más apto de P_{nuevo} añadir $\vec{\hat{x}}$ a P_{nuevo}
// Avanzar generación
22 $P \leftarrow P_{\text{nuevo}}$
23 **regresa** ($\vec{x} \in P$ tal que $A(\vec{x}) \geq A(\vec{y}), \forall \vec{y} \in P \setminus \{\vec{x}\}$) // el individuo más apto

Notar que recibe como entrada el tamaño de la población, las probabilidades de cruza y mutación, además de la función de aptitud. Valores típicos para las probabilidades de cruza y mutación son 0.60 a 0.95 y 0.001 a 0.01, respectivamente.

De acuerdo a lo anterior, el algoritmo genético tiene las siguientes características:

- Hace uso de selección probabilística.
- Normalmente emplea codificación binaria, es decir, cadenas de 1s y 0s.
- Se evoluciona el genotipo y no el fenotipo, este último solo se usa para la función de aptitud y para obtener un valor numérico.
- La cruza es el operador principal mientras que la mutación es un operador secundario.
- El elitismo es un operador indispensable para garantizar convergencia.

3.3. Estrategias Evolutivas

Las estrategias evolutivas son una clase de heurística de naturaleza aleatoria para búsqueda directa en espacios continuos introducidas por Rechenberg y Schwefel en Alemania a mediados de los 1960s [25]. A diferencia de los AGs, en una estrategia evolutiva se evoluciona el fenotipo y no el genotipo de los individuos. A continuación se describe la implementación más simple [25]:

1. El problema se plantea como encontrar el vector D-dimensional de números reales $\vec{x}$ que optimiza la función objetivo $f : \mathbb{R}^D \mapsto \mathbb{R}$. Sin pérdida de generalidad, supongamos que deseamos minimizar a f.
2. Una población inicial de vectores padres, $\vec{x}_i$, $i = 1, ..., \mu$, es generada de forma aleatoria sobre el subespacio de $\mathbb{R}^D$ considerado como la región factible (recordar que el problema puede estar sujeto a restricciones). La distribución de los puntos iniciales es normalmente uniforme.
3. Un vector hijo , $\vec{x}'_i$, $i = 1, ..., \mu$, es creado a partir de cada vector padre al perturbar cada componente de este último con una variable aleatoria Gaussiana con media cero y desviación estándar σ preseleccionada. Este procedimiento corresponde a la mutación.
4. En la selección se determinan cuáles de todos los vectores (padres e hijos) se conservan para que representen a la siguiente generación de vectores padres. La selección es con base en su aptitud (por ejemplo, aptitud$_i = -f(\vec{x}_i)$). En este sentido se eligen los μ vectores con mayor aptitud del conjunto.

5. El proceso de generar nuevos vectores de prueba (hijos) y seleccionar aquellos con mejor aptitud continúa hasta que se cumple la condición de paro.

Bajo este contexto, cada componente de una posible solución $\vec{x}$ es visto como un rasgo de comportamiento y no como un gen [25]. Por tanto, la evolución es a nivel del comportamiento del individuo y no a nivel genético. A continuación se muestra el algoritmo de una $(\mu + \lambda) - EE$ que describe los 5 pasos antes mencionados:

Algoritmo 3.2: Estrategia Evolutiva, variante $(\mu + \lambda) - EE$ donde $\lambda = \mu$

entrada: μ, σ, A
salida : max A

1 Generar una población M de forma aleatoria con distribución uniforme compuesta por μ individuos
2 Para cada $\vec{x}_i \in M$, calcular $A(\vec{x}_i)$ // A es la función de aptitud
3 **mientras** no se cumpla el criterio de paro **hacer**
4 **para** cada $\vec{x}_i \in M$ **hacer**
5 $\vec{x}'_i = \vec{x}_i + N(0, \sigma)$ // $N(0,\sigma)$ es un vector de números Gaussianos independientes
6 $H^i = \vec{x}'_i$ // conjunto de vectores de prueba (hijos)
7 Para cada $\vec{x}_i \in H$, calcular $A(\vec{x}_i)$
8 $M \leftarrow M \cup H$
9 Conservar a los μ individuos más aptos de M y eliminar al resto
10 **regresa** ($\vec{x} \in M$ tal que $A(\vec{x}) \geq A(\vec{y}), \forall \vec{y} \in M \setminus \{\vec{x}\}$) // el individuo más apto

Como se puede observar, el operador de mutación (al que nos referimos como el operador principal) es el único que se emplea en la variante de EE presentada. Sin embargo, eso no significa que no exista el operador de cruza (en el contexto de EEs se denomina recombinación). Esto se debe a que existen múltiples variantes, unas hacen uso sólo del operador principal y otras hacen uso de ambos operadores (mutación y recombinación).

Todas las variantes pueden ser definidas de acuerdo a la siguiente notación:

$$(\mu/\rho \overset{+}{,} \lambda) - EE,$$

donde μ corresponde al tamaño de la población, λ al número de hijos creados por generación y ρ a el número de padres que se usan para procrear a un hijo. El término $\overset{+}{,}$ se refiere a la estrategia de selección adoptada:

- **Selección "+":** si λ hijos son creados por generación entonces los μ individuos que avanzan a la siguiente generación son los más aptos entre la unión del conjunto de hijos y el conjunto de padres [27].

- **Selección ",":** el conjunto de padres se descarta automáticamente y de los λ hijos creados se eligen a los μ más aptos para que avancen a la siguiente generación. Por tanto, se pide que $\lambda > \mu$ [27].

Lo que distingue a las variantes que usan recombinación de las que no es el valor que adquiere ρ:

- **Caso 1:** si $\rho = 1$ significa que se emplea un padre para crear a un hijo. En este caso, se entiende que no hay recombinación como tal. Dichas variantes son conocidas como $(\mu + \lambda) - EE$ y $(\mu, \lambda) - EE$ [27].

- **Caso 2:** si $\rho > 1$ significa que al menos se usan dos padres para crear a un hijo. En este caso, sí hay recombinación [27].

A continuación se describe el procedimiento que sigue una EE en general:

Algoritmo 3.3: $(\mu/\rho \overset{+}{,} \lambda) - EE$

entrada: μ, ρ, λ, σ, A, sel
salida : max A

1 Generar una población M de forma aleatoria con distribución uniforme compuesta por μ individuos
2 Para cada $\vec{x}_i \in M$, calcular $A(\vec{x}_i)$ // A es la función de aptitud
3 **mientras** no se cumpla el criterio de paro **hacer**
4 **para** $k = 0$ hasta $k < \lambda$ **hacer**
5 $P \leftarrow$ seleccionar ρ vectores de forma aleatoria de la población M // conjunto de padres
6 $\vec{r} =$ recombinar(P, ρ) // emplear alguna estrategia de recombinación
7 $\vec{x}' =$ mutar$(\vec{r}, \sigma)$ // ver algoritmo 3.2, linea 5
8 $H^k = \vec{x}'$ // conjunto de vectores de prueba (hijos)
9 Para cada $\vec{x}_k \in H$, calcular $A(\vec{x}_k)$
10 **si** $sel == 0$ **entonces**
  // Usar selección "+"
11 $M \leftarrow M \cup H$
12 Conservar a los μ individuos más aptos de M y eliminar al resto
13 **otro**
  // Usar selección ","
14 Conservar a los μ individuos más aptos de H y eliminar al resto
15 $M \leftarrow H$
16 **regresa** ($\vec{x} \in M$ tal que $A(\vec{x}) \geq A(\vec{y}), \forall \vec{y} \in M \setminus \{\vec{x}\}$) // el individuo más apto

El procedimiento es bastante claro tomando en cuenta que se parece mucho al presentado en el algoritmo 3.2. El operador de mutación que se emplea es el mismo al presentado en el algoritmo 3.2, línea 5. Respecto al operador de cruza, existen tres tipos de recombinación [27]:

1. **Recombinación por clonación:** cuando $\rho = 1$ significa que se usa un padre para procrear a un hijo. En este caso, se emplea la clonación, es decir, el hijo es una copia exacta de su padre.

2. **Recombinación discreta:** el vector $\vec{r}$ se define de la siguiente forma:

- Para cada componente r_j de $\vec{r}$ hacer:
- Generar un número entero n aleatoriamente con distribución uniforme en el intervalo $[0, \rho-1]$. Todos los padres del conjunto P tienen la misma probabilidad de ser seleccionados.
- Tomar al n-ésimo vector padre $\vec{q}_n$ de P.
- Definir a r_j como el j-ésimo componente del n-ésimo padre $q_{n,j}$, es decir, $r_j = q_{n,j}$.

Normalmente, al padre seleccionado se le denomina "dominante"; de ahí que a este tipo de recombinación también se le conozca como "recombinación dominante".

3. **Recombinación intermedia:** consiste en simplemente calcular el centro de masa (centroide) de los ρ padres en P:

$$r_j = \frac{1}{\rho} \sum_{n=0}^{\rho-1} q_{n,j} \tag{3.2}$$

Este tipo de recombinación sólo está definida para espacios continuos.

Para aquellos lectores interesados en una versión adaptativa del procedimiento presentado en el algoritmo 3.3 se sugiere revisar [27].

De acuerdo a lo presentado, la estrategia evolutiva tiene las siguientes características:

- Selección determinística.
- Comúnmente empleada para realizar búsquedas en espacios continuos.
- Se evoluciona el fenotipo y no el genotipo.
- La mutación es el operador principal mientras que la cruza es un operador secundario.

3.4. Programación Evolutiva

La programación evolutiva (PE) fue propuesta por Fogel, Owens y Walsh en los 1960s [17]. El enfoque de esta técnica es muy similar al de las EEs, aunque la programación evolutiva se propuso originalmente para resolver problemas de predicción.

La idea es evolucionar un algoritmo que opera sobre un conjunto de símbolos vistos hasta el momento y que son producidos por un cierto ambiente de tal forma que se produzca una salida que maximice el beneficio del algoritmo en espera del siguiente símbolo. Se decidió que los algoritmos serían representados como máquinas de estados finitos (MEFs)

La PE opera sobre MEFs de la siguiente forma [25]:

1. Se inicializa una población de máquinas padres de forma aleatoria (cada padre o individuo de la población es una MEF codificada).

2. Los padres se exponen al ambiente, es decir, se les alimenta con el conjunto de símbolos que han sido generados hasta ahora. Para cada máquina padre, conforme cada símbolo de entrada es alimentado, cada símbolo de salida es comparado con el siguiente símbolo de entrada. El valor de esta predicción es luego medido por una función de beneficio. Después de llevarse a cabo la última predicción, la aptitud de la máquina es asignada empleando una función de aptitud que recibe como entrada el beneficio logrado por cada símbolo.

3. Máquinas hijas se crean al mutar cada máquina padre de forma aleatoria. Hay cinco posibles tipos de mutación que naturalmente surgen a partir de la definición de MEF: cambiar un símbolo de salida, cambiar una transición de estado, agregar un estado, eliminar un estado o cambiar el estado inicial. El borrado de un estado y el cambio del estado inicial son permitidos únicamente cuando la máquina tiene más de un estado. Las mutaciones son llevadas a cabo respecto a una distribución de probabilidad, la cual normalmente es uniforme.

4. Las máquinas recién creadas son evaluadas con el conjunto de símbolos vistos hasta el momento de la misma forma como fueron evaluados los padres.

5. Las máquinas hijas que implican un mayor beneficio pasan a la siguiente generación. Normalmente el tamaño de la población se mantiene constante.

6. Los pasos del 3 al 5 se repiten de forma iterativa hasta que se requiera realizar una predicción del siguiente símbolo (nunca antes visto) proporcionado por el ambiente. La mejor solución (máquina) es seleccionada para encargarse de esta predicción. El nuevo símbolo se agrega al conjunto de símbolos vistos hasta el momento, y el proceso salta al paso 2.

La PE también puede ser aplicada a problemas de optimización sobre espacios continuos aunque de cierta forma es equivalente en muchos casos a algunos procedimientos usados en las EEs. A pesar de esto, es posible resaltar dos puntos importantes que distinguen a ambas técnicas [25]:

- Las EEs hacen uso de selección estrictamente determinística. La PE usualmente se basa en la selección probabilística ya que suele emplear una selección tipo torneo estocástico para definir qué individuos pasan a la siguiente generación. La probabilidad de que cierto sujeto sobreviva depende de su rango (el cual es función de su aptitud).

- Las EEs suponen que los individuos dentro de una población son de la misma especie por lo que es posible aplicar el operador de cruza. Por su parte, la PE supone que los

individuos que componen una población pertenecen a especies distintas por lo que el operador de cruza no es aplicable, ya que especies distintas no pueden recombinarse entre sí.

3.5. Programación Genética

El término "programación genética" se refiere a todo el conjunto de algoritmos evolutivos que generan programas [14]. La técnica de PG se consolidó en el área de cómputo evolutivo debido al trabajo hecho por John Koza a finales de los 1980s. La PG sigue un flujo de operación similar al de los AGs además de emplear el mismo concepto de codificación y decodificación de los individuos. La codificación de un individuo (en el contexto de PG un individuo corresponde a un programa) es referido como su genoma o genotipo el cual, en el caso de los AGs, es representado con una estructura lineal de datos. En un principio, la PG también empleaba una codificación lineal. Sin embargo, años después Koza formalizó la idea de usar un árbol como estructura de datos para representar el genoma de un individuo [14].

Un programa puede ser representado por un árbol tal como lo hacen los compiladores: leen el código fuente de un programa, lo dividen en tokens, analizan estos tokens y finalmente crean un árbol de sintaxis abstracta (ASA). En la figura 3.4 se muestra el ejemplo de un ASA para el sencillo procedimiento que se muestra en el algoritmo 3.4. Los nodos internos de un ASA contienen operadores los cuales pueden ser distintas funciones, y los nodos terminales u hojas contienen los operandos.

Algoritmo 3.4: Imprimir números pares.

entrada: rango (debe ser un número natural)
salida : $\{x \in [0, \text{rango}] \mid x \text{ es par}\}$

```
1 i = rango
2 mientras i ≥ 0 hacer
3 |   si i mod 2 == 0 entonces
4 |   |   imprimir i
5 |   i = i − 1
```

El funcionamiento básico de la PG es muy similar al de un AG. Su implementación es como sigue [17]:

1. **Generar de forma aleatoria una población inicial de individuos y asignar aptitud**. En los AGs se crea un conjunto de cadenas de bits de forma aleatoria. Para la PG se hace los mismo pero en lugar de dichas secuencias unidimensionales se crean árboles. Normalmente, hay una profundidad máxima l_{max} especificada que los árboles no pueden violar, por lo que la distancia desde la raíz hasta la hoja más lejana de los árboles que sean creados no es mayor que l_{max}. Existen varios métodos de inicialización:

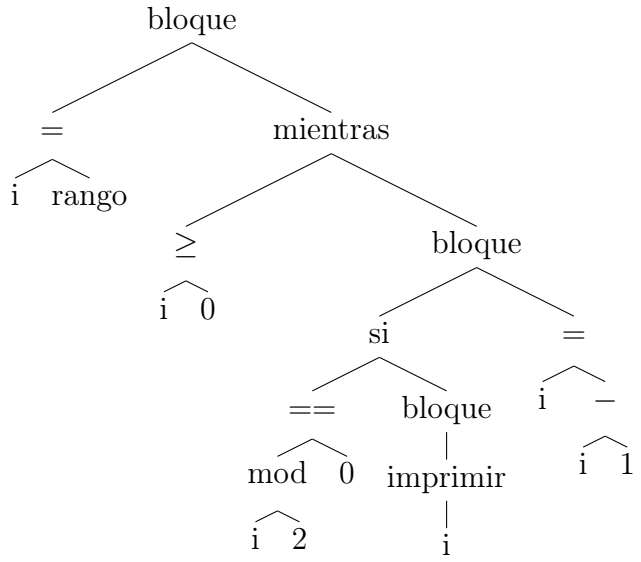

Figura 3.4: Árbol de sintaxis abstracta para el algoritmo 3.4.

- **Por crecimiento (*grow method*)**: se crean árboles donde cada camino desde la raíz a las hojas no es más largo que l_{max} pero puede ser más corto. Esto se logra al decidir de forma aleatoria para cada nodo si debe ser una hoja o no cuando es anexado al árbol [14].
- **Por completitud (*full method*)**: se crean árboles donde cada camino desde la raíz hasta las hojas tiene exactamente la longitud l_{max} [14].
- **Por rampa media y media (*ramped half-and-half*)**: combina los dos métodos anteriores: la mitad de la población se inicializa usando el método por crecimiento, y la otra mitad se inicializa usando el método por completitud. Sin embargo, para garantizar una mayor diversidad, en cada mitad se utilizan valores distintos del parámetro de tamaño máximo l_{max} para cada individuo [17].

2. **Asignar aptitud a cada individuo de la población**. Esto se hace igual que en los AGs.

3. **Selección, cruza y mutación**. Primero se crea un conjunto de parejas empleando algún método de selección (individuos con mayor aptitud tiene más posibilidad de ser seleccionados). Luego, se aplica el operador de cruza para cada pareja generando así dos nuevos individuos de los cuales se toma al más apto y se elimina al otro. Finalmente, se aplica el operador de mutación al único hijo y se agrega a la siguiente generación.
   La cruza es el operador principal en la PG al igual que en los AGs. Se emplea un

método de recombinación que es análogo a la cruza de un punto en AGs. Consiste en seleccionar un subárbol al azar de cada uno de los padres para posteriormente intercambiarlos justo en sus respectivas posiciones seleccionadas (ver figura 3.5).
La mutación es un operador secundario el cual simplemente consiste en seleccionar de forma aleatoria un nodo del árbol y substituir el subárbol enraizado en ese nodo con otro generado aleatoriamente.

4. **Avanzar generación y repetir**. El conjunto de hijos avanza a la siguiente generación. Se cuida que el mejor de la siguiente generación sea, al menos, tan apto como el mejor de la generación pasada (aplicar elitismo puede ser útil para garantizar esta condición). Repetir los pasos del 2 al 4 hasta que se cumpla el criterio de paro.

5. **Presentar al individuo más apto de la población**.

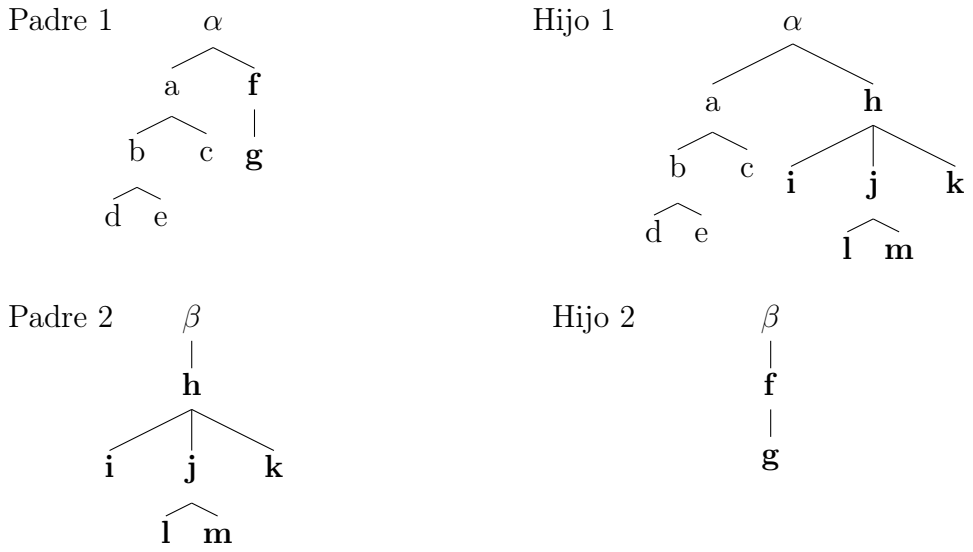

Figura 3.5: Cruza por intercambio de subárboles

De acuerdo a lo visto anteriormente, la programación genética presenta las siguientes características:

- Hace uso de selección probabilística.
- Normalmente emplea codificación arbórea.
- Se evoluciona el genotipo y no el fenotipo.
- Se emplea principalmente para la generación de programas.
- La cruza es el operador principal mientras que la mutación es un operador secundario.

4 | Evolución Diferencial

La evolución diferencial (ED) surgió como un algoritmo evolutivo muy competitivo en optimización continua en la década de los 1990s. El primer trabajo publicado sobre la ED apareció como un reporte técnico elaborado por R. Storn y K. V. Price en 1995. En poco tiempo, la comunidad de cómputo evolutivo se dio cuenta que la ED mostraba un desempeño sobresaliente en múltiples clases de problemas de optimización sobre espacios continuos en comparación al resto de los algoritmos evolutivos [1].

El modo en que opera la ED es similar al de otros algoritmos evolutivos (AG, EE, PE y PG): (1) crear una población inicial y (2) aplicar los operadores de selección, cruza y mutación de forma iterativa hasta que se cumpla la condición de paro.

Así mismo, la ED es considerada como una hermana cercana de la EE ya que tienen varias características en común: (1) incorporan selección determinística, (2) se usan principalmente para realizar búsquedas en espacios continuos, (3) evolucionan el fenotipo y (4) la mutación es el operador principal.

La ED ha sido aplicada a múltiples problemas de optimización en diversas áreas de ciencia e ingeniería; su popularidad se debe a que es simple y fácil de implementar, su complejidad espacial es baja en comparación con algunos de los optimizadores globales para espacios continuos más competitivos, su número de parámetros de control es muy pequeño y suele tener un buen rendimiento [1].

A pesar de las ventajas de la ED, este algoritmo también es víctima de la maldición de la dimensionalidad al igual que el resto de los AEs; su desempeño se deteriora conforme aumenta el número de variables de decisión. No obstante, los investigadores continúan haciendo esfuerzos para mejorar su rendimiento en presencia de alta dimensionalidad.

En este capítulo se discute detalladamente el algoritmo de evolución diferencial resaltando su uso en problemas de optimización global de un solo objetivo. En la sección 4.1 se presenta la estructura general de la ED. En la sección 4.2 se introduce el esquema originalmente presentado por R. Storn y K. V. Price. En la sección 4.3 se presentan los diferentes esquemas que existen para el algoritmo de la ED así como la nomenclatura para identificarlos. En la sección 4.4 se discuten configuraciones adecuadas para los parámetros de control y se presentan algunos mecanismos que existen en la literatura para ajustarlos automáticamente. Finalmente, en la sección 4.5 se presentan las ideas más relevantes en la literatura que se han propuesto para mejorar el desempeño de la ED en problemas de

optimización global a gran escala.

4.1. Introducción

El algoritmo de evolución diferencial exhibe, desde una perspectiva general, un flujo de operación tal como se muestra en el algoritmo 4.1. Es posible notar que además de la función objetivo f, la ED requiere de sólo tres parámetros de control para su funcionamiento [1, 28]:

1. $\boldsymbol{NP}$: corresponde al número de individuos que componen a una población P, es decir, el tamaño de la población. Este parámetro es un número entero positivo mayor o igual a cuatro.

2. $\boldsymbol{F}$: corresponde al factor de escalamiento el cual es empleado durante el procedimiento de mutación. Este parámetro es un número real que normalmente es definido en el intervalo $[0, 2]$.

3. $\boldsymbol{Cr}$: corresponde a la tasa de recombinación empleada durante el procedimiento de cruza (en el contexto de la ED se le denomina recombinación). Este parámetro es un número real que representa una probabilidad, por lo que se define dentro del intervalo $[0, 1]$.

Algoritmo 4.1: Evolución Diferencial

entrada: NP , Cr , F, f
salida : min $f(\vec{x})$ tal que $\vec{x} \in \Omega$

1 Crear una población P de forma aleatoria y uniformemente distribuida sobre Ω // $NP =$ tamaño de la población
2 **mientras** el criterio de paro no se cumpla **hacer**
3 **para** $i = 0$ to $i < NP$ **hacer** // para cada individuo de la población
  // Aplicar el operador de mutación: recibe como entrada a la población actual y el factor de escalamiento F
4 $\vec{v}_i \leftarrow$ mutación(P, F)
  // Aplicar operador de recombinación: recibe como entrada al individuo en curso x_i y al vector mutante en curso v_i. También requiere la tasa de recombinación Cr.
5 $\vec{u}_i \leftarrow$ recombinación$(\vec{x}_i, \vec{v}_i, Cr)$
  // Operador de selección
6 $P^i_{\text{nuevo}} \leftarrow$ el mejor entre $\vec{u}_i$ y $\vec{x}_i$ // selección: tomar $\vec{u}_i$ si $f(\vec{u}_i) \leq f(\vec{x}_i)$, tomar $\vec{x}_i$ de lo contrario
7 $P \leftarrow P_{\text{nuevo}}$ // avanzar generación y repetir
8 **regresa** $(\vec{x} \in P$ tal que $f(\vec{x}) \leq f(\vec{y}), \forall \vec{y} \in P \setminus \{\vec{x}\})$ // el mejor: $\vec{x}_{best}$

El primer paso consiste en crear aleatoriamente una población de individuos dentro de la región factible Ω (ver algoritmo 4.1, línea 1). Por simplicidad se supondrá, de aquí en adelante, que el problema de optimización presenta sólo restricciones de límite. En este caso, la j-ésima componente del i-ésimo individuo se inicializa como sigue:

$$x_{i,j} = lb_j + (ub_j - lb_j) * \text{UniformRand}(0, 1)$$

donde lb_j y ub_j representan el límite inferior y superior de la j-ésima variable, respectivamente. Cuando un individuo es generado, se evalúa su valor con la función objetivo (notar que no se emplea una función de aptitud como tal sino que se usa a f directamente como indicador de calidad). Una vez finalizada la fase de inicialización, se repiten los pasos de mutación, recombinación y selección para cada individuo de la población (líneas 2-7) mientras el criterio de detención no se cumpla.

El operador de mutación (línea 4) hace uso de un esquema o **estrategia de mutación** para generar a un vector mutante $\vec{\boldsymbol{v}}$ (conocido como el **vector donante**).

El operador de recombinación (ver línea 5) se emplea para genera a un vector hijo $\vec{\boldsymbol{u}}$ (también conocido como el **vector de prueba**) usando como padres a $\vec{v}$ y al individuo en curso $\vec{\boldsymbol{x}}$, este último denominado el **vector objetivo**.

Finalmente, el operador de selección se usa para elegir simplemente al mejor entre los vectores objetivo (padre) y prueba (hijo) para que represente al vector objetivo de la siguiente generación (ver línea 6). Notar cómo en este caso un sujeto es mejor que otro cuando su valor, al evaluarlo con f, es menor (esto se debe a que estamos suponiendo un problema de minimización).

Respecto a los procedimientos de mutación y recombinación, existe más de una forma de llevarlos a cabo. Actualmente, existen varias estrategias de mutación y unos pocos tipos de recombinación. En el contexto de la ED, diferentes procedimientos de mutación y recombinación dan lugar a diferentes esquemas de evolución.

4.2. El esquema clásico

Normalmente se identifica al esquema clásico de la ED como aquel presentado originalmente por R. Storn y K. V. Price en 1995 (ver [28]). La estrategia de mutación y el tipo de recombinación que incorpora se describen a continuación. Así mismo se describe con mayor detalle el procedimiento de selección.

Mutación

El procedimiento de mutación consiste en elegir aleatoriamente (distribución uniforme) tres individuos $\vec{x}_{r_1}, \vec{x}_{r_2}, \vec{x}_{r_3} \in P$ tal que sean distintos entre ellos y al individuo que se está procesando en ese momento (es decir, $r_1 \neq r_2 \neq r_3 \neq i$). Después se define a $\vec{v}_i$ como:

$$\vec{v_i} = \vec{x}_{r_1} + F * (\vec{x}_{r_2} - \vec{x}_{r_3})$$

donde al vector $\vec{x}_{r_1}$ se le conoce como **vector base**. Este procedimiento o estrategia de mutación puede llevarse a cabo siempre y cuando $|P| \geq 4$, es decir, debe existir una población con al menos cuatro miembros [28].

La ED difiere notablemente de algoritmos como la EE y la PE en cuanto a que se mutan (perturban) a los vectores base (padres secundarios) con diferencias vectoriales escaladas

derivadas de la población. A medida que transcurren las generaciones, estas diferencias tienden a adaptarse a la escala natural del problema. Por ejemplo, si la población se compacta en una variable pero permanece ampliamente dispersa en otra, los vectores de diferencia muestreados de ella serán pequeños en la primera variable, pero grandes en la última. Esta adaptación automática mejora significativamente la convergencia del algoritmo. En otras palabras, la EE y PE requieren la especificación o adaptación del tamaño de paso absoluto para cada variable durante generaciones, mientras que la ED solo requiere la especificación de un único factor de escala relativo (F) para todas las variables. [1]

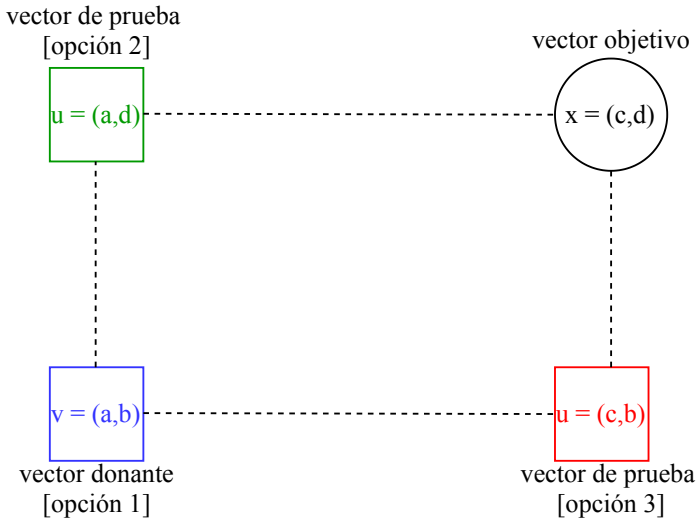

Figura 4.1: Posibles resultados para el vector de prueba usando recombinación binomial.

Recombinación

El procedimiento de recombinación consiste en generar aleatoriamente (distribución uniforme) un número entero j_{rand} en el intervalo $[0, D-1]$ (suponiendo que la indización de arreglos comienza en 0). Luego, por cada entrada j del vector $\vec{u}$, se genera un número real r' de forma aleatoria (distribución uniforme) en el intervalo $[0, 1]$. Si r' es menor o igual a Cr o j es igual a j_{rand} entonces $\vec{u}$ hereda su j-ésimo componente de $\vec{v}$; de lo contrario, lo hereda de $\vec{x}$.

Este tipo de recombinación se conoce como **binomial**. La idea de generar a j_{rand} en un principio es para garantizar que al menos una de las componentes sea heredada de $\vec{v}$. Si Cr toma un valor por debajo de 0.5, entonces se espera que la mayoría de las componentes sean heredadas de $\vec{x}$ y si toma un valor por arriba de 0.5 se espera lo opuesto [1].

En la figura 4.1 se muestra un ejemplo de los posibles resultados para el vector de prueba al recombinar $\vec{v}$ y $\vec{x}$ en un espacio de dos dimensiones. Se sabe que al menos una componente de $\vec{u}$ es heredada de $\vec{v}$ por lo que $\vec{u}$ no puede ser igual a $\vec{x}$ en ningún caso.

Sin embargo, puede ser posible que $\vec{u}$ sea igual a $\vec{v}$ sobre todo si $Cr > 0.5$ (ver figura 4.1, opción 1). Por otro lado, es posible que $\vec{x}$ herede cualquiera de sus dos componentes (0 o 1) a $\vec{u}$ sobre todo si $Cr < 0.5$. En este caso, es posible obtener dos casos más (ver figura 4.1, opciones 2 y 3).

La ED, al igual que la EE, emplea el operador de recombinación para crear un único vector de prueba, mientras que la mayoría de los AGs recombinan dos vectores padre para producir dos vectores de prueba (hijos). Nótese que el procedimiento de recombinación binomial es prácticamente una cruza uniforme o discreta la cual también puede ser implementada en una EE y un AG además de la ED [1].

Selección

La selección puede ser aplicada a un AE principalmente en dos etapas diferentes: la primera etapa es la selección de los padres para decidir qué vectores de la población actual se recombinarán, mientras que la segunda es la selección de sobrevivientes, es decir, elegir qué vectores de las poblaciones de padres y descendientes avanzarán a la próxima generación.

- **Selección de padres.** A diferencia de los AGs que seleccionan a los padres en función de su aptitud, tanto la EE como la ED brindan a todos los individuos la misma oportunidad de ser seleccionados como padres. Por ejemplo, en la EE cada individuo tiene la misma posibilidad de ser seleccionado para mutación (y recombinación). Por su parte en la ED la selección de padres es totalmente determinista (ver algoritmo 4.1, línea 5) [1].

- **Selección de sobrevivientes.** Respecto a la selección para supervivencia, existen situaciones en donde sólo se permite avanzar a los vectores descendientes (como se hace en algunos AGs simples). En un escenario como éste, no hay garantía de que la mejor solución hasta ese momento se conserve. Retener a la mejor solución vista hasta el momento se llama elitismo y juega un papel importante para que el algoritmo pueda converger al óptimo global. Por esta razón la mayoría de los AEs, incluidos la ED, la PE y algunas variantes de las EEs, toman en cuenta a la población actual para determinar la inclusión de los vectores descendientes a la siguiente generación. La ED se caracteriza por llevar a cabo un procedimiento de selección de supervivencia tipo torneo binario entre vectores objetivo (padres) y sus vectores de prueba correspondientes (hijos) donde en una generación (iteración completa) se llevan a cabo NP contiendas. Dicho procedimiento de selección garantiza afinar a la mejor solución de la generación pasada o al menos conservar una tan buena como ella (implícitamente incorpora el operador de elitismo). Otra ventaja importante es que se mantiene un buen nivel de diversidad en la población ya que las élites y sus descendientes no pueden dominar a la población [1].

El esquema descrito es conocido en la literatura como $DE/rand/1/bin$ donde DE se refiere a evolución diferencial por sus siglas en inglés, *rand* y *1* se refieren a la acción de elegir al vector base de forma aleatoria y usar una diferencia vectorial para perturbarlo (durante el procedimiento de mutación) respectivamente. Finalmente, el término *bin* alude al hecho de que se usa recombinación binomial.

En el algoritmo 4.2 se muestra el procedimiento completo para el esquema $DE/rand/1/bin$.

Algoritmo 4.2: $DE/rand/1/bin$

entrada: NP , Cr , F, f
salida : min $f(\vec{x})$ tal que $\vec{x} \in \Omega$

Crear una población P de forma aleatoria y uniformemente distribuida sobre Ω // NP = tamaño de la población
mientras el criterio de paro no se cumpla **hacer**
  para $i = 0$ hasta $i < NP$ **hacer** // para cada individuo de la población
    // Mutación
    Tomar $r_1, r_2, r_3 \in [0, NP-1]$ de forma aleatoria // $r_1 \neq r_2 \neq r_3 \neq i$, $r_n \in \mathbb{N}$
    $\vec{v}_i = \vec{x}_{r_1} + F * (\vec{x}_{r_2} - \vec{x}_{r_3})$ // $F \in [0.0, 2.0]$ se refiere al factor del escalamiento, $\vec{x}_{r_n} \in P$
    // Recombinación
    $j_{rand} = \text{urnd}(\text{int}, 0, D-1)$ // $j_{\text{rand}} \in \mathbb{N}$
    para $j = 0$ to $j < D$ **hacer** // para cada componente, aplicando recombinación binomial
      si $\text{urnd}(\text{real}, 0, 1) \leq Cr$ || $j == j_{\text{rand}}$ **entonces** // $Cr \in [0.0, 1.0]$ se refiere a la tasa de recombinación
        $u_{i,j} = v_{i,j}$ // heredar del vector donante (resultado de la mutación)
      otro
        $u_{i,j} = x_{i,j}$ // heredar del vector objetivo (padre)
    // Selección
    $P^i_{\text{nuevo}} \leftarrow$ el mejor entre $\vec{u}_i$ y $\vec{x}_i$ // selección: tomar $\vec{u_i}$ si $f(\vec{u_i}) \leq f(\vec{x_i})$, tomar $\vec{x_i}$ de lo contrario
  $P \leftarrow P_{\text{nuevo}}$ // avanzar generación y repetir
regresa $(\vec{x} \in P$ tal que $f(\vec{x}) \leq f(\vec{y}), \forall \vec{y} \in P \setminus \{\vec{x}\})$ // el mejor: $\vec{x}_{\text{best}}$

4.3. Los diferentes esquemas

En el trabajo realizado por R. Storn y K. V. Price además de presentarse el denominado esquema clásico ($DE/rand/1/bin$), también se introdujeron varios esquemas alternativos. La principal razón de presentar esquemas alternativos se debe a que algunos de ellos pueden ser útiles en problemas en los que otros no, ya que el comportamiento de la búsqueda es sensible al esquema adoptado. Como puede intuirse, lo que distingue a un esquema de otro es la estrategia de mutación y el tipo de recombinación que se incorpora, pero como sólo existen dos tipos de recombinación, binomial (bin) y exponencial (exp), la estrategia de mutación es realmente la que distingue un esquema de otro [1].

Por ejemplo, consideremos el esquema clásico: la estrategia de mutación que incorpora básicamente perturba a un vector base (el cual es reseleccionado aleatoriamente cada vez que es invocado el operador de mutación) con la diferencia vectorial de dos vectores tomados aleatoriamente de P. Lo anterior resulta en vectores mutantes dispersos cerca

de los vectores base correspondientes [29] (ver figura 4.2). Suponiendo que se tiene a la mano un mecanismo aleatorio de reelección altamente uniforme, se esperaría que en una generación todos los vectores de la población actual (o casi todos) fueran elegidos como vectores base al menos una vez, permitiendo así a esta estrategia ser de tipo explorativo (brindar mayor diversidad al conjunto de vectores mutantes).

En la práctica, el mecanismo de reelección aleatorio puede incurrir en la sobrealeatoriedad, lo que puede reducir, en cierto modo, la capacidad explorativa del algoritmo [29]. Esto no significa que dicha estrategia sea inútil, aunque si lo que se desea es promover una mayor diversidad en el conjunto de vectores mutantes se puede adoptar la siguiente estrategia:

$$\vec{v}_i = \vec{x}_i + F * (\vec{x}_{r_2} - \vec{x}_{r_3}),$$

donde ahora el vector base es justamente el vector objetivo. El esquema que incorpora dicha estrategia de mutación y recombinación *bin* corresponde a $DE/target/1/bin$.

Ahora, supongase por un momento que no existe interés en dispersar los vectores mutantes cerca de los vectores objetivo, sino más bien en dispersarlos cerca del **mejor** vector objetivo (denominado $\vec{x}_{best}$). En este caso se puede adoptar la siguiente estrategia:

$$\vec{v}_i = \vec{x}_{best} + F * (\vec{x}_{r_2} - \vec{x}_{r_3}).$$

Aunque dicha estrategia puede inducir a la pérdida de diversidad y presentar problemas de convergencia prematura [29], hay escenarios donde puede ser útil, sobre todo en contextos donde existe interés en que los miembros del conjunto de vectores mutantes estén localizados en la vecindad del mejor candidato (búsqueda local). El esquema que incorpora dicha estrategia de mutación corresponde a $DE/best/1/bin$.

En la literatura existe una nomenclatura especial para distinguir a los diferentes esquemas de la ED. La sintaxis que se usa para tal fin es la siguiente:

$$DE/\alpha/\beta/\gamma,$$

donde α especifica la elección del vector base, β representa el número de diferencias vectoriales empleadas para perturbar al vector base y γ se refiere al tipo de recombinación empleada.

Como puede observarse, un cambio en el esquema de mutación puede impactar el sentido de la búsqueda de distinta forma. A continuación se presentan algunas otras estrategias de mutación que existen [1]:

- $DE/pbest/1/\gamma$: el término *pbest* se refiere a un individuo de la población que es reelegido de forma aleatoria (cada vez que se invoca el operador de mutación) donde la elección se hace entre los $\lfloor p * NP \rfloor$ mejores candidatos, siendo $p \in (0.0, 1.0]$ un parámetro ajustable

$$\vec{v}_i = \vec{x}_{pbest} + F * (\vec{x}_{r_2} - \vec{x}_{r_3})$$

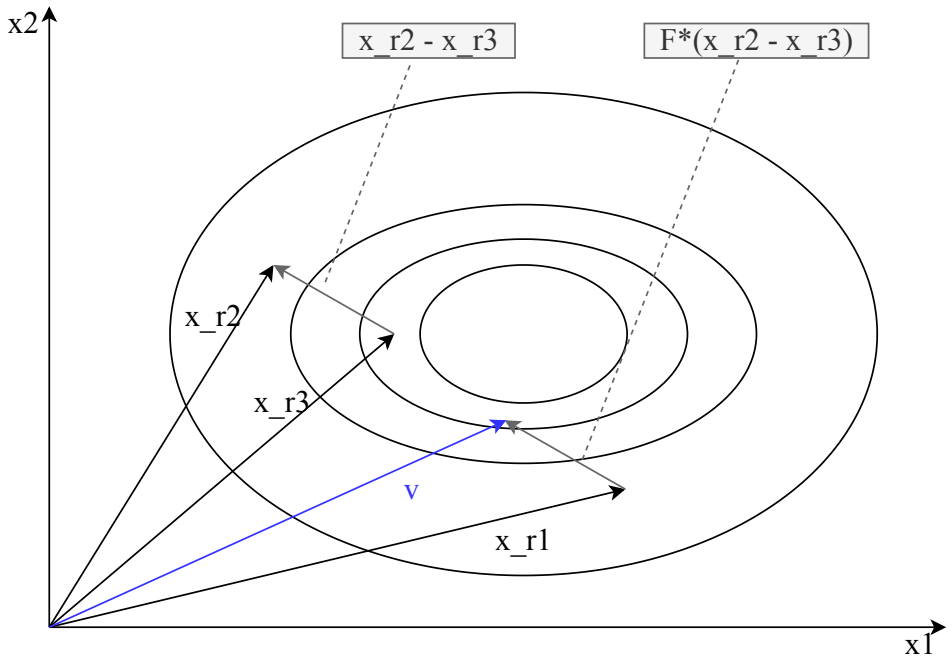

Figura 4.2: Ilustración del esquema estándar de mutación en un espacio bidimensional

- $DE/target-to-best/1/\gamma$:

$$\vec{v}_i = \vec{x}_i + F * (\vec{x}_{best} - \vec{x}_i) + F * (\vec{x}_{r_2} - \vec{x}_{r_3})$$

- $DE/best/2/\gamma$:

$$\vec{v}_i = \vec{x}_{best} + F * (\vec{x}_{r1} - \vec{x}_{r2}) + F * (\vec{x}_{r_3} - \vec{x}_{r_4})$$

- $DE/rand/2/\gamma$:

$$\vec{v}_i = \vec{x}_{r1} + F * (\vec{x}_{r2} - \vec{x}_{r3}) + F * (\vec{x}_{r_4} - \vec{x}_{r_5})$$

Los índices r_1, r_2, r_3, r_4 y r_5 son enteros mutuamente excluyentes generados de forma aleatoria.

En realidad existen diversas estrategias de mutación que van desde sencillas modificaciones de las ya presentadas hasta otras más elaboradas como lo es la mutación basada en el centro [18], la mutación trigonométrica [1] o la mutación triangular [7].

Respecto a la recombinación sólo existen, como se mencionó, dos tipos de operadores de este tipo: binomial y exponencial (ver algoritmo 4.3). Generalmente, se emplea la recombinación binomial mientras que la exponencial es poco común.

Considerando que se han presentado 7 distintas estrategias de mutación (incluyendo la clásica) y que cada una de ellas puede ser combinada con uno de los dos tipos de recombinación, es posible enumerar al menos 14 diferentes esquemas de la ED. Para los lectores

interesados en más esquemas de la ED, se sugiere revisar [1].

Adicionalmente, existen versiones multiestrategia de la ED que se caracterizan por adoptar un conjunto o grupo de estrategias de mutación con el fin de elegir la más conveniente acorde a la circunstancia; la forma de elegir (entre las diferentes estrategias) varía entre algoritmos. Por ejemplo, en EDE [29] se consideran 2 estrategias de mutación y se elige una de ellas en función del estado de la búsqueda; al inicio de la búsqueda la estrategia 1 tiene más probabilidad de ser seleccionada, mientras que la estrategia 2 incrementa su probabilidad de ser elegida hacia el final de la búsqueda.

En SaDE [30] se consideran 4 estrategias diferentes: en cada generación a cada individuo se le asigna aleatoriamente una estrategia de mutación diferente (en la generación 0 todas tienen la misma probabilidad de ser asignadas). Al finalizar una generación, la tasa de éxito (generar hijos más aptos que sus respectivos padres) y fracaso asociada a cada estrategia se registra en una memoria. En las generaciones subsecuentes, las estrategias con mayor tasa de éxitos adquieren mayor probabilidad de ser seleccionadas [1].

Otras propuestas que emplean un grupo de estrategias de mutación son CoDE [31] y EPSDE [32].

Normalmente a las versiones de la ED que incluyen múltiples estrategias de mutación, nuevos operadores ajenos a los presentados, mecanismos de ajuste de parámetros y otras extensiones son identificadas como variantes de la ED. Por ejemplo, los algoritmos mencionados (EDE, SaDE, CoDE y EPSDE) son variantes ya que su funcionamiento está basado en la ED pero añaden nuevas mecánicas (con el fin de mejorar el desempeño).

Algoritmo 4.3: Recombinación exponencial.

entrada: $\vec{v}_i$, $\vec{x}_i$, Cr
salida : $\vec{u}_i$

$j_{\text{rand}} = \text{urnd}(\text{int}, 0, D-1)$ // D → dimension del problema
$\vec{u}_i \leftarrow \vec{x}_i$ // hacer a $\vec{u}_i$ igual a $\vec{x}_i$
$L = 0$
$j = j_{\text{rand}}$
hacer
  $u_{i,j} = v_{i,j}$
  $j = (j+1)\,\%\, D$ // % representa el operador módulo
  $L++$
mientras $(\text{urnd}(\text{real}, 0, 1) \leq Cr$ y $L < D)$
regresa $\vec{u}_i$

4.4. Parámetros de control

Como se indicó en las sección 4.1, el algoritmo de la ED requiere la definición de los siguientes parámetros de control para su funcionamiento: factor de escalamiento F, tasa de recombinación Cr y tamaño de la población NP.

Originalmente, Storn y Price indicaron (ver [28]) que un valor razonable para NP estaba entre $5D$ y $10D$ (donde D corresponde a la dimensionalidad del problema) y que el

rango efectivo de F estaba en el intervalo $[0.4, 1]$, siendo 0.5 una buena elección inicial. En caso de que la población convergiera prematuramente, entonces F o NP deberían incrementarse.

Como una buena elección inicial, el valor sugerido para Cr es 0.1; se indica que valores de Cr cercanos a 1.0 regularmente aceleran la convergencia, así que intentar $Cr = 0.9$ o $Cr = 1.0$ es recomendable en un primer intento con el fin de investigar si es posible obtener una solución rápida.

Es importante mencionar que la configuración de parámetros sugerida por Storn y Price es respecto al esquema clásico de la ED. Lo anterior es un inconveniente ya que una misma configuración puede no ser adecuada bajo diferentes esquemas. Otro problema es que que la configuración de parámetros de control también es dependiente del problema [33], es decir, puede ser que una cierta configuración funcione bien para un problema A pero no para un problema B.

De acuerdo con estudios más recientes (ver [1]), la mayoría de los investigadores indican que un valor adecuado para F está en el intervalo $[0.4, 0.95]$ con $F = 0.9$ como una buena elección inicial. Por su parte, un valor adecuado para Cr está en el intervalo $[0.0, 0.2]$ cuando la función objetivo es separable y $[0.9, 1.0]$ cuando los parámetros de dicha función son dependientes, es decir, hay un cierto grado de no separabilidad.

La realidad es que no existen suficientes justificaciones experimentales para definir un rango adecuado de los parámetros de control, especialmente F y Cr, además de que la interacción entre la configuración establecida y el rendimiento de optimización sigue siendo complicada y no se comprende completamente [1, 34]. En consecuencia, se comenzó a considerar la idea de auto-ajuste de parámetros con el fin de determinar automáticamente una configuración acorde al problema y a la etapa de evolución [1, 34].

4.4.1. Ajuste automático

Los mecanismos de adaptación de parámetros (MAPs) que existen en la literatura generalmente están diseñados para ajustar los parámetros de control F y Cr; la mayoría de ellos pueden ser categorizados con base en cómo llevan a cabo el ajuste [34]. A continuación se describen las principales clases de MAPs que existen [34]:

1. **Control determinista de parámetros (clase D)**: el parámetro de control es alterado por algunas reglas deterministas sin tener en cuenta retroalimentación de la búsqueda evolutiva.

2. **Control adaptativo de parámetros (clase A)**: se emplea retroalimentación de la búsqueda evolutiva para cambiar dinámicamente los parámetros de control.

3. **Control de parámetros auto-adaptativos (clase sA)**: se utiliza un enfoque del tipo "la evolución de la evolución" para llevar a cabo la auto-adaptación de

los parámetros de control. Los parámetros de control están directamente asociados con los individuos por lo que experimentan mutación y recombinación. Dado que los mejores valores de los parámetros tienden a generar individuos que tienen más probabilidades de sobrevivir, estos valores pueden propagarse a más descendientes.

La mayoría de los MAPs siguen un enfoque acorde a las clases A y sA. Si dichos mecanismos están bien diseñados pueden mejorar la robustez de la ED adaptando dinámicamente los parámetros a las características de los diferentes paisajes asociados a las funciones objetivo. Por lo tanto, es aplicable a varios problemas de optimización sin la necesidad de ajustar (prueba y error) manualmente los parámetros. Además, la tasa de convergencia puede mejorarse si los parámetros de control se adaptan a los valores apropiados en diferentes etapas de la evolución de un problema en particular [34].

A continuación se introduce una variante adaptativa y otra auto-adaptativa de la ED con el fin de ejemplificar los enfoques asociados a las clases A y sA.

- **jDE** [35]: es una variante auto-adaptativa de la ED. Los parámetros de control F y Cr son codificados en el individuo, es decir, se extiende el vector de variables de tal forma que dichos parámetros de control también estén sujetos a optimización además de las variables de decisión del problema. La idea es que los mejores valores de estos parámetros de control codificados conduzcan a mejores individuos que, a su vez, tengan más probabilidades de sobrevivir y producir descendencia [1]. En la generación 0, los parámetros de control $F_{i,G=0}$ y $Cr_{i,G=0}$ asociados al individuo i se inicializan en 0.5 y 0.9, respectivamente [34]. Posteriormente, los nuevos parámetros de control correspondientes a la siguiente generación ($F_{i,G+1}$ y $Cr_{i,G+1}$) se calculan de acuerdo con la siguiente regla [35, 1]:

$$F_{i,G+1} = \begin{cases} (F_l + rnd_1) * F_u & \text{con probabilidad } \tau_1 \\ F_{i,G} & \text{de lo contrario} \end{cases}$$

$$Cr_{i,G+1} = \begin{cases} rnd_2 & \text{con probabilidad } \tau_2 \\ Cr_{i,G} & \text{de lo contrario} \end{cases}$$

  donde F_l y F_u son los límites inferior y superior de F, respectivamente. Nótese que rnd_1 y rnd_2, son números reales generados aleatoriamente en el intervalo $[0, 1]$, mientras que τ_1 y τ_2 son nuevos parámetros que se añaden al sistema. Normalmente las variantes auto-adaptativas y adaptativas de la ED incluyen parámetros extra. No obstante, esto es justificable ya que suelen dar mejores resultados que los esquemas tradicionales de la ED, principalmente porque se sustituyen parámetros sensibles por otros menos sensibles [1]. En el algoritmo de jDE se definen: $\tau_1 = \tau_2 = 0.1$, $F_l = 0.1$ y $F_u = 0.9$. Es importante mencionar que $F_{i,G+1}$ y $Cr_{i,G+1}$ son generados previo al procedimiento de mutación de la generación $G + 1$. Resultados experimentales sugieren que jDE se desempeña bastante mejor que el esquema clásico de la ED [34].

- **JADE** [34]: es una variante adaptativa de la ED. Cada individuo $\vec{x}_i$ está asociado a sus valores $F_{i,G}$ y $Cr_{i,G}$, pero a diferencia de como sucede con jDE, dichos parámetros no son codificados como dos variables más de decisión. La idea del enfoque adaptativo es estimar los nuevos valores $F_{i,G+1}$ y $Cr_{i,G+1}$ con base en el historial de éxito, es decir, en valores que exitosamente dieron lugar a mejores individuos. En la generación 0, los parámetros de control $F_{i,G=0}$ y $Cr_{i,G=0}$ se inicializan ambos en 0.5. Posteriormente, los nuevos parámetros de control correspondientes a la siguiente generación se calculan de acuerdo con la siguiente expresión:

$$\begin{aligned} F_{i,G+1} &= \text{Cauchy}(\mu_F, 0.1) \\ Cr_{i,G+1} &= \text{Normal}(\mu_{Cr}, 0.1) \end{aligned} \tag{4.1}$$

  donde los nuevos valores de F y Cr son generados usando las distribuciones Cauchy y Normal empleando μ_F y μ_{Cr} como medias, respectivamente. Notar que en ambos casos se usa una desviación estándar de 0.1. En caso de que $Cr_{i,G+1}$ esté fuera del intervalo $[0, 1]$ se lleva a cabo un procedimiento de corrección que consiste en truncarlo (0 o 1) dependiendo de si viola el límite inferior o el superior. Por otro lado, si $F_{i,G+1} > 1$ se trunca a 1. Por el contrario, si $F_{i,G+1} \leq 0$, entonces se repite $F_{i,G+1} = \text{Cauchy}(\mu_F, 0.1)$ hasta que se genere un valor válido [33].
  Los valores μ_F y μ_{Cr} son estimados a partir del conjunto de valores que ayudaron a producir hijos más aptos que sus padres. Cuando en una generación G se detecta que $\vec{u}_i$ es mejor que $\vec{x}_i$ inmediatamente se almacenan los valores $F_{i,G+1}$ y $Cr_{i,G+1}$ en una memoria especial. Al finalizar la generación G, se calcula el promedio de los valores registrados y dichos valores medios estimados son empleados al comienzo de la siguiente generación para generar nuevos valores de F y Cr (ver ecuación 4.1). Es importante mencionar que la memoria donde se almacena los valores de interés se reinicia al final de cada generación.

El enfoque adaptativo es el más popular, sobre todo en problemas donde el número de variables de decisión es elevado; un MAP que adopta el enfoque auto-adaptativo es menos efectivo que uno que adopta el enfoque adaptativo en problemas a gran escala ya que el rendimiento del proceso de optimización se deteriora conforme aumenta el espacio de búsqueda.

Otras variantes adaptativas de la ED son EPSDE [32], CoDE [31], SaDE [30] y SHADE [33].

4.5. Evolución diferencial en problemas de optimización global a gran escala mono-objetivo

El algoritmo de evolución diferencial ha mostrado ser efectivo en múltiples clases de problemas de optimización global. No obstante, es bien sabido que su rendimiento se

deteriora conforme se incrementa la dimensionalidad del problema. Normalmente, la ED muestra una excelente capacidad de búsqueda cuando se aplica a problemas con 30 a 100 variables y usualmente su desempeño se deteriora cuando la dimensionalidad del espacio de búsqueda va más allá de 500 [1]. Dado que actualmente los problemas de optimización global a gran escala (OGGE) más desafiantes tienen al menos 1000 variables de decisión, los esquemas tradicionales de la ED están bastante limitados en este contexto.

Actualmente, existen diversas propuestas algorítmicas en el área de OGGE basadas totalmente o parcialmente en la ED:

- **Algoritmos totalmente basados en la ED.** Son variantes de la ED que añaden nuevos mecanismos pensados especialmente para amortiguar las deficiencias que presenta en su forma básica.

- **Algoritmos parcialmente basados en la ED.** Son propuestas que emplean a la ED como un componente del algoritmo. Usualmente la combinan con otros métodos para complementar el proceso de búsqueda. Este tipo de propuestas no son consideradas variantes de la ED como tal.

Tal como se discutió en el capítulo 2, las propuestas para OGGE pueden seguir dos enfoques distintos: (1) descomposición y (2) no descomposición. Los algoritmos basados en descomposición adoptan el marco de coevolución cooperativa (CC) el cual consiste en dividir el problema original en subproblemas y luego resolver cada uno por separado. Por el contrario, los no basados en descomposición atacan el problema como un todo.

Acorde con lo anterior, los algoritmos destinados a OGGE que hacen uso de la ED pueden ser clasificados tal como se muestra en el mapa conceptual de la figura 4.3. Los algoritmos CC pueden emplear el mismo optimizador para resolver cada uno de los subcomponentes (por ejemplo, emplear la ED para resolver todos los subproblemas) o pueden emplear diferentes optimizadores (por ejemplo, se tienen 4 subcomponentes y se decide resolver dos de ellos con la ED y el resto con una EE). Es importante mencionar que cuando se escribe "emplear la ED" nos referimos a cualquier esquema o variante de ella. Decidir cuál o cuáles optimizadores emplear para resolver los subcomponentes no es realmente tan importante debido a que el rendimiento de un algoritmo CC es más sensible a la técnica de descomposición que a la resolución de los subproblemas mismos.

Los algoritmos no basados en descomposición pueden ser variantes de la ED o la ED combinada con otro métodos (algoritmos híbridos).

A continuación se describe brevemente un conjunto de algoritmos representativos en el área de OGGE que hacen uso de la ED. Dichas propuestas son descritas con el fin de exponer las ideas fundamentales que llevaron a su desarrollo y para ejemplificar el uso de la ED en problemas de OGGE. Se comienza abordando los algoritmos basados en coevolución cooperativa, posteriormente se describen los basados en hibridación y finalmente se habla algunas de las variantes de la ED que han sido desarrolladas para OGGE.

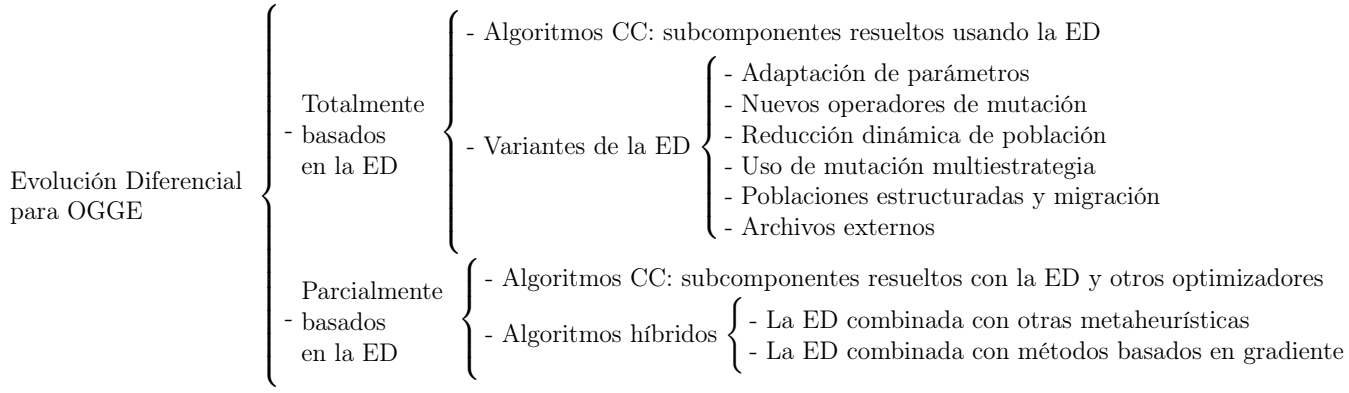

Figura 4.3: Evolución diferencial en problemas de optimización global a gran escala monoobjetivo.

4.5.1. Evolución diferencial con coevolución cooperativa

DECC-G

El algoritmo de evolución diferencial con coevolución cooperativa usando agrupación aleatoria , DECC-G [9], fue una de las primeras propuestas en emplear a la ED adoptando un enfoque basado en descomposición [4]; la idea es descomponer un problema de optimización a gran escala en un conjunto de subproblemas más pequeños que el original, es decir, descomponer un vector D-dimensional (asociado a una solución del problema) en m (donde $m << D$) subcomponentes s-dimensionales empleando agrupación aleatoria

Una de las novedades que incorporó en su momento fue la técnica de agrupación aleatoria la cual se caracteriza por otorgar a cada variable la misma probabilidad de ser asignada a cualquiera de los m subcomponentes. La motivación detrás de esto es aumentar la posibilidad de optimizar las variables que interactúan juntas [4].

La resolución de los subproblemas es llevada a cabo por la ED donde cada subcomponente se resuelve por separado. Un aspecto interesante del algoritmo de DECC-G es que emplea una variante adaptativa de la ED como optimizador.

Cuando se cumple el criterio de paro y termina el algoritmo, este ensambla los mejores individuos de cada subpoblación o especie para regresar el vector solución. Cabe señalar que la estructura de agrupación cambia dinámicamente, es decir, el procedimiento de agrupación aleatoria es invocado previo al inicio de cada generación [4].

Otra novedad que incorporó DECC-G fue una estrategia de ponderación adaptativa. Idealmente, cada subcomponente debería consistir de variables que interactúan estrechamente mientras que la interacción entre los subcomponentes debería ser débil, justamente el mecanismo de ponderación trata de controlar esto [9].

MLCC

Con el fin de mejorar el algoritmo de DECC-G se propuso un nuevo esquema denominado coevolución cooperativa multi-nivel (MLCC [36]) para problemas de optimización a gran escala.

El algoritmo de DECC-G está sujeto a un parámetro difícil de determinar (una de sus principales desventajas) denominado tamaño de grupo (variables a optimizar por subcomponente). Dicho parámetro tiene un gran impacto en el rendimiento del algoritmo ya que su valor depende del problema. Por ejemplo, un tamaño de grupo pequeño es útil para problemas separables, mientras que uno grande es adecuado para problemas no separables [4]. Para tratar dicho inconveniente se propuso emplear el algoritmo de DECC-G pero adoptando el entorno MLCC lo cual resultó en el algoritmo de DEMLCC-G que a diferencia de DECC-G incorpora un mecanismo adaptativo para "ajustar" el tamaño de grupo de acuerdo con las características del problema y el estado de la evolución [4].

En el algoritmo de DEMLCC-G se considera un conjunto de estrategias de descomposición con diferentes tamaños de grupo tal que cada una representa diferentes niveles de interacción entre variables [4]. Al comienzo de cada generación se selecciona aleatoriamente una estrategia de descomposición (en un principio todas tienen la misma probabilidad de ser elegidas) y posteriormente se lleva a cabo el procedimiento de agrupación aleatoria acorde a la estrategia de descomposición seleccionada. Después se lleva a cabo el procedimiento de optimización (resolución de subproblemas), se calcula y actualiza la tasa de crecimiento del rendimiento tomando en cuenta la estrategia de descomposición seleccionada y el ciclo se repite. Las estrategias de descomposición que ayudan a registrar mayores tasas de crecimiento tienen más probabilidad de ser elegidas en las siguientes generaciones [4].

DECC-DG

EL algoritmo de DECC-DG opera acorde a la estructura típica de un algoritmo CC pero adoptando una estrategia de descomposición denominada agrupación diferencial (ver [13]).

A diferencia de los algoritmos de DECC-G y MLCC que emplean la técnica de agrupación aleatoria sin previo conocimiento de los problemas, el algoritmo de DECC-DG descompone los problemas de optimización a gran escala mediante la técnica de agrupación diferencial la cual se caracteriza por detectar la interacción entre las variables de decisión permitiendo que las que interactúan se asignen a los mismos subcomponentes manteniendo así la interdependencia entre ellos al mínimo [4].

En realidad, el algoritmo de DECC-DG es muy similar a los algoritmos DECC-G y DEMLCC-G, excepto que en el primero se usa la técnica de agrupación diferencial y que ésta última se lleva a cabo una única vez, antes de comenzar el proceso evolutivo, es decir, durante la etapa de inicialización [4]. Es importante mencionar que la técnica de

agrupación diferencial es bastante más competitiva que la de agrupación aleatoria [4].

4.5.2. Evolución diferencial combinada con otros optimizadores

MOS

La propuesta denominada "muestreo múltiple de descendientes" (MOS [3], por sus siglas en inglés) es un algoritmo híbrido no basado en descomposición y considerado una de las mejores metaheurísticas para OGGE por varios años [1] [6]. El algoritmo de MOS incorpora un marco de trabajo que permite combinar diferentes algoritmos siguiendo un enfoque de relevo híbrido a alto nivel (HRH, por sus siglas en inglés). En este caso, dos o más algoritmos son ejecutados uno tras otro, cada uno de ellos reutilizando la salida del anterior.

El algoritmo de MOS está compuesto por ocho algoritmos; cinco de ellos son AEs y el resto son metaheurísticas de búsqueda local. Dentro del conjunto de AEs se tiene un AG y cuatro variantes de la ED. El proceso de búsqueda es dividido en diferentes pasos (bloques de un número fijo de evaluaciones de la función objetivo) de tal forma que en cada paso se ejecutan todos los algoritmos secuencialmente. La participación (proporción del número de evaluaciones en un paso) de cada algoritmo para el paso $i + 1$ se ajusta de acuerdo con su rendimiento en el paso anterior i de tal forma que los algoritmos con mejores rendimientos tendrán mayor participación en los siguientes pasos.

Resultados experimentales (ver [3]) indican que MOS obtiene resultados estadísticamente superiores que el algoritmo de DECC-G en la mayoría de los problemas de prueba (se emplea el conjunto de prueba más reciente para OGGE denominado *CEC'13 LSGO benchmark* [37]).

MLSHADE-SPA

La propuesta denominada evolución diferencial basada en el historial de éxito con reducción lineal del tamaño de la población y adaptación de semi-parámetros (MLSHADE-SPA [7], por sus siglas en inglés) es un algoritmo coevolutivo cooperativo, híbrido y uno de los dos primeros en demostrar ser más competitivo que el algoritmo de MOS en la mayoría de problemas del conjunto de prueba *CEC'13 LSGO* [38].

El algoritmo de MLSHADE-SPA, al igual que el de MOS, incorpora un marco híbrido combinando algoritmos basados en población y buscadores locales. En este caso, la propuesta está compuesta por 4 algoritmos; tres variantes de las ED y un método de búsqueda local (el buscador local 1 del algoritmo de búsqueda de trayectoria múltiple, MTS-LS1 [19] por sus siglas en inglés).

El proceso de optimización es dividido en múltiples rondas o iteraciones (se usan 50 ron-

[1] De acuerdo con los estándares de las competencias de optimización global a gran escala llevadas a cabo durante los congresos de cómputo evolutivo.

das) donde se adopta un número máximo de evaluaciones de la función objetivo (max_{FEs}) como criterio de paro. Por tanto, se dispone de $\frac{max_{FEs}}{50}$ evaluaciones por ronda. Cada ronda, a su vez, se compone de tres fases:

1. **Exploración:** se optimiza el problema como un todo usando una de las tres variantes de la ED y el 25 % de los recursos (número de evaluaciones de la función objetivo) asignados a la ronda.

2. **Exploración usando coevolución cooperativa:** las dimensiones del problema son divididas e integradas en tres subcomponentes empleando agrupación aleatoria donde cada subcomponente es optimizado empleando cada una de las tres variantes de la ED. Esta fase consume el 25 % de los recursos asignados a la ronda.

3. **Explotación:** se ensamblan de nuevo las dimensiones y se optimiza el problema como un todo empleando el buscador local MTS-LS1. Esta fase consume el 50 % de los recursos asignados a la ronda.

Para más detalles sobre el algoritmo de MLSHADE-SPA se sugiere revisar [7]. Actualmente esta propuesta es un algoritmo de referencia en el área de OGGE [39].

SHADE-ILS

La propuesta denominada evolución diferencial basada en el historial de éxito con búsqueda local iterativa (SHADE-ILS [6], por sus siglas en inglés) es un algoritmo no basado en descomposición, híbrido y el segundo en demostrar ser más competitivo que el algoritmo de MOS en la mayoría de problemas del conjunto de prueba *CEC'13 LSGO* [38, 6]. En el algoritmo de SHADE-ILS, al igual que en MLSHADE-SPA, el proceso de búsqueda está compuesto por múltiples iteraciones o rondas de tal forma que en cada iteración se lleva a cabo una fase explorativa y otra explotativa.

El algoritmo que se emplea durante la fase explorativa es SHADE el cual es una variante de la ED que incorpora un mecanismo adaptativo de parámetros bastante sólido [33]. La fase explotativa es manejada por los algoritmos de búsqueda local MTS-LS1 y L-BFGS-B (ver [40]). En una iteración se selecciona aleatoriamente uno de los dos algoritmos de búsqueda local donde en un principio la probabilidad de selección es igual para ambos, sin embargo esto cambia de acuerdo al desempeño registrado en iteraciones pasadas. Un aspecto importante de resaltar es el comportamiento de búsqueda complementario de los buscadores locales adoptados; el algoritmo de MTS-LS1 es una metaheurística y por ende un método no basado en gradiente mientras que el algoritmo de L-BFGS-B es un método exacto que usa información sobre el gradiente (en realidad una aproximación a él) para guiar la búsqueda. De acuerdo con los autores, el algoritmo de MTS-LS1 es un método bastante rápido y apropiado para problemas separables pero es muy sensible a rotaciones mientras que el algoritmo de L-BFGS-B es menos poderoso pero también es menos sensible a rotaciones.

El algoritmo de SHADE-ILS implementa varias de las ideas adoptadas en los algoritmos de MOS y MLSHADE-SPA tal como como combinar diferentes algoritmos, dividir el proceso de búsqueda en exploración y explotación iterativamente y añadir un mecanismo para ajustar el algoritmo (buscador local) más adecuado acorde al problema y al estado de evolución. Adicionalmente, el algoritmo de SHADE-ILS incorpora un mecanismo especial de reinicio el cual es activado cuando se detecta un estancamiento (atrapado en un óptimo local). Actualmente, SHADE-ILS es un algoritmo de referencia en el área de OGGE y uno de los mejores algoritmos híbridos no basados en descomposición.[39, 6].

4.5.3. Variantes de evolución diferencial para OGGE

LMDEa

El algoritmo de evolución diferencial con detección de modalidades de paisaje y un archivo de diversidad (conocido en la literatura como LMDEa [10]) es una variante de la ED especialmente diseñada para problemas de OGGE. Esta propuesta plantea atender varios de los incovenientes que presenta la ED al trabajar con problemas de alta dimensionalidad:

- No se puede usar un tamaño de población muy grande debido el límite de recursos.
- Es difícil mantener la diversidad en la población si no hay suficientes individuos en ella.
- El valor de los parámetros de control F y Cr son dependientes del problema.

EL algoritmo de LMDEa fue diseñado para funcionar bien con un número pequeño de individuos en la población (por ejemplo, los experimentos fueron llevados a cabo con $NP = 60$), también incorpora un archivo externo A para mantener una mayor diversidad en la población (A está constituido por vectores de prueba que fueron derrotados en generaciones previas) y un mecanismo de detección del tipo de paisaje de aptitud el cual es capaz de determinar si éste es unimodal o multimodal con el fin de ajustar a F según corresponda.

Al comienzo de cada generación se determina si es momento de activar el mecanismo de detección del tipo de paisaje de aptitud (la frecuencia de activación T_d es un parámetro definido por el usuario). De ser así, entonces se ejecuta la rutina y se actualiza F. Luego, se llevan a cabo los procedimientos de mutación, recombinación y selección para cada individuo i de la población. La estrategia de mutación empleada es la misma adoptada en el esquema clásico de la ED con la diferencia de que ahora el vector $\vec{x}_{r3}$ es seleccionado aleatoriamente del conjunto $P \cup A$. Respecto al procedimiento de recombinación, se utiliza el tipo *bin* pero en caso de que el vector de prueba generado no sea mejor que el vector objetivo entonces se repiten los procedimientos de mutación y recombinación pero ahora empleando recombinación *exp*. Finalmente, los vectores de prueba derrotados son añadidos al archivo externo A. Es importante mencionar que el valor de Cr_i es generado

aleatoriamente en el intervalo $[0.8, 1]$ en caso de usar recombinación *bin* y en el intervalo $[0, 1]$ en caso de emplear recombinación *exp*.

El algoritmo de LMDEa fue puesto a prueba con los algoritmos de DECC-G y DEMLCC-G empleando el conjunto de prueba *CEC'10 LSGO* [41]. Los resultados indicaron (ver [10]) que el algoritmo de LMDEa es, en promedio, mejor que DECC-G y DEMLCC-G en 15 de los 20 problemas de prueba. Así mismo, de acuerdo con los resultados y pruebas realizadas (ver [7]) se indica que el algoritmo de MLSHADE-SPA presenta un mejor desempeño, en general, que el algoritmo de LMDEa (empleando el conjunto de prueba *CEC'10 LSGO*).

EADE

El algoritmo de evolución diferencial adaptativa mejorada (EADE [11], por sus siglas en inglés) es una variante de la ED originalmente propuesta para trabajar con problemas de OGGE. Entre sus principales características se tiene una nueva estrategia de mutación y un nuevo mecanismo para adaptar el parámetro de control Cr.

La nueva estrategia de mutación que se propone en el algoritmo de EADE usa dos vectores seleccionados aleatoriamente de entre los mejores y peores $\lfloor NP * p \rfloor$ individuos de la población, respectivamente, mientras que un tercer vector es seleccionado de forma aleatoria de entre los individuos que están en la mitad. Es decir, aquellos que no están en el conjunto de los peores y mejores. En este caso, el valor del parámetro $p \in (0, 1)$ es definido por el usuario. El esquema de mutación es como sigue:

$$\vec{v}_i = \vec{x}_r + F_1 * (\vec{x}_{p\_best} - \vec{x}_r) + F_2 * (\vec{x}_r - \vec{x}_{p\_worst})),$$

donde $\vec{x}_{p\_best}$ y $\vec{x}_{p\_worst}$ son los vectores seleccionados aleatoriamente de los conjuntos mejores y peores, respectivamente. $\vec{x}_r$ es un vector seleccionado aleatoriamente no perteneciente ni al conjunto de los mejores ni al de los peores, mientras que F_1 y F_2 son factores de escalamiento independientemente generados para cada individuo de la población acorde con una distribución uniforme en el intervalo $(0, 1)$. De acuerdo con los autores, esta estrategia de mutación ayuda a mantener efectivamente el equilibrio entre la exploración global y las capacidades de explotación local durante el proceso de búsqueda de la ED.

El mecanismo adaptativo de Cr es algo elaborado pero la idea general es disponer de un conjunto o grupo con diferentes valores los cuales cambian de forma gradual de acuerdo con experiencias pasadas (para mayores detalles se sugiere revisar [11]). Es importante mencionar que en el algoritmo de EADE cada individuo tiene su propia configuración de valores F_{1_i}, F_{2_i} y Cr_i.

El algoritmo de EADE además de usar la nueva estrategia de mutación también incorpora la estrategia clásica donde la forma de decidir cual de ellas emplear (para cada individuo de la población) es aleatoria con una probabilidad de 0.5, es decir, se lanza una moneda; si cae cara entonces se usa la estrategia nueva, de lo contrario, se usa la clásica.

De acuerdo con los resultados experimentales (ver [11]), el algoritmo de EADE presen-

ta un desempeño similar al de LMDEa (no se encontró una diferencia estadísticamente significativa entre ellos empleando el conjunto de prueba *CEC'10 LSGO*).

4.5.4. Observaciones finales

Se han presentado varios algoritmos representativos en el área de OGGE con el fin de exhibir la mayoría de los diferentes enfoques implementados hasta el momento.

El enfoque más popular y sencillo es dividir el problema en otros más pequeños y resolverlos por separado, de esta forma es posible reducir un problema de gran escala a muchos problemas de mediana e incluso baja escala lo cual es ventajoso ya que la ED normalmente puede manejar situaciones así. Otra ventaja de este enfoque es que puede ser altamente paralelizable además de escalable. En general, los métodos de descomposición son adecuados, pero su principal desventaja es que: **no todos los problemas son separables**. Establecer la estrategia de descomposición adecuada no es un problema sencillo [13] y en realidad el buen desempeño de un algoritmo basado en coevolución cooperativa depende de ello tal como se puede observar de los tres métodos CC presentados. La diferencia significativa entre ellos reside básicamente en el procedimiento de descomposición.

El enfoque híbrido es una idea bastante razonable ya que cada algoritmo puede ser útil en distintos tipos de problemas. Para que el desempeño general de un algoritmo híbrido mejore es importante usar métodos que se complementen correctamente y es por ello que el proceso de optimización regularmente es dividido en épocas o iteraciones las cuales a su vez dividen la búsqueda en exploración y explotación (dos actividades complementarias). El algoritmo de ED es bastante popular como componente explorativo dado que es simple, su complejidad espacial así como la temporal es aceptable, exhibe un buen desempeño en problemas de optimización global, existen múltiples formas de adaptarle y es fácilmente modificable. El componente explotativo regularmente es un método de búsqueda local no basado en población el cual puede ser una metaheurística o un método basado en gradiente (tal como se vio en el algoritmo de SHADE-ILS). La ventaja de las propuestas híbridas no basadas en descomposición respecto a las de CC es que no degradan su desempeño tan drásticamente en problemas donde hay interdependencia entre variables. Sin embargo, tienen tres desventajas importantes: **es más complicado escalarlas**, **son menos paralelizables** (principalmente porque los métodos de búsqueda local son difícilmente paralelizables y generan cuellos de botella en su procesamiento) y **el tiempo de cómputo es más elevado** (ya que se está resolviendo el problema como un todo).

Finalmente, las variantes de la ED destinadas a resolver problemas de OGGE son construidas a partir de modificaciones y extensiones de la estructura base de la ED. En realidad, este tipo de propuestas presentan las mismas desventajas que los algoritmos híbridos, aunque son más paralelizables ya que regularmente no integran procedimientos de búsqueda local. Es importante mencionar que normalmente este tipo de propuesta suelen ser superadas por algoritmos híbridos y de CC.

5 | Esquema Propuesto

Actualmente, existen diversas propuestas algorítmicas de optimizadores evolutivos (OEs) para resolver problemas de optimización global a gran escala siendo el enfoque basado en descomposición el más empleado entre ellos. No obstante, también existen OEs que atacan los problemas como un todo. Estos OEs han demostrado ser competitivos respecto con aquellos basados en descomposición sobre todo en problemas donde las variables de decisión están altamente correlacionadas (problemas no separables).

Los OEs que siguen un enfoque no basado en descomposición normalmente adoptan un marco híbrido ya que su estructura compuesta por múltiples métodos permite mayor adaptabilidad. Es decir, es más probable que puedan resolver un problema sin previo conocimiento sobre sus propiedades (separable, no-separable, parcialmente separable, multimodal, unimodal, etc.).

En este capítulo, se presenta una nueva propuesta con un enfoque que no usa descomposición y que está totalmente basada en el algoritmo de evolución diferencial y especialmente diseñada para optimización global a gran escala. El diseño retoma varias de las ideas que incorporan algunos de los algoritmos híbridos representativos del área, en particular la estrategia de búsqueda: **global-local** [1]. No obstante, dicho diseño no explota la idea de hibridación realmente, sino más bien la versatilidad de la ED para combinar diferentes esquemas de evolución y estructuras poblacionales.

En la sección 5.1 se describe el diseño de la nueva propuesta y se introduce el nuevo algoritmo denominado GL-SHADE. En la sección 5.2 se presenta y describe a detalle el algoritmo de SHADE ya que éste es uno de los componentes fundamentales de la propuesta. En la sección 5.3 se introduce y describe con detalle el algoritmo de $\text{eSHADE}_{\text{ls}}$ el cual también corresponde a uno de los principales componentes de la propuesta. En la sección 5.4 se discuten brevemente los procedimientos de corrección sugeridos para el manejo de restricciones de límite de las variables de decisión al trabajar con el algoritmo de evolución diferencial (o variantes de ella) cuando un problema de optimización global mono-objetivo está sujeto a dichas restricciones. Finalmente, en la sección 5.5 se discuten detalles de la implementación de la nueva propuesta.

[1] Un proceso de búsqueda que realiza exploración y explotación de forma iterativa.

5.1. Diseño

El diseño que a continuación se expone surge como una propuesta para responder a la pregunta: ¿Es posible construir un algoritmo basado en la evolución diferencial que iguale o supere el desempeño de los mejores algoritmos híbridos propuestos hasta ahora para OGGE?

A la fecha, las propuestas basadas en la ED generalmente son superadas por sus contrapartes híbridas principalmente por su incapacidad de establecer un equilibrio entre exploración y explotación. Los algoritmos híbridos tratan este problema al combinar métodos de búsqueda global y local iterativamente tal como se muestra en la figura 5.1.

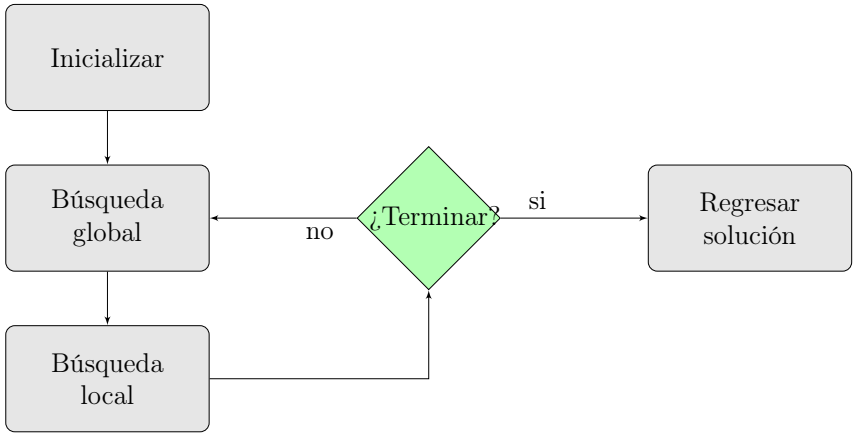

Figura 5.1: Estrategia de búsqueda global-local.

Las propuestas basadas en la ED para problemas de OGGE normalmente intentan realizar exploración y explotación empleando una población y un esquema evolutivo pensado para cubrir ambos tipos de búsqueda. No obstante, en este caso se propone adoptar la estrategia de búsqueda global-local estableciendo un esquema estructurado de dos poblaciones. La población 1 incorpora un esquema evolutivo especializado en búsqueda global mientras que la población 2 adopta uno especializado en búsqueda local. En el sentido estricto de la palabra, las poblaciones en este caso son más bien subpoblaciones o *demes* ya que es necesario que los individuos de ambos grupos puedan comunicarse e incluso reproducirse. La comunicación entre las subpoblaciones es mediante el operador de migración usando una de las estrategias más sencillas la cual consiste en sólo enviar o recibir al mejor individuo hacia o desde la otra subpoblación, respectivamente. Dado que en este caso se adopta un enfoque de no descomposición y la estrategia de búsqueda de la figura 5.1, no es posible evolucionar ambas poblaciones al mismo tiempo (en paralelo) aunque si es posible paralelizar el proceso evolutivo de una población.

Acorde con lo anterior, en el algoritmo 5.1 se muestra el pseudocódigo correspondiente al diseño que se plantea el cual se ha denominado búsqueda global y local empleando evo-

lución diferencial (GL-DE). En este caso, además del problema de optimización se requiere definir tres parámetros de entrada correspondientes al número máximo de evaluaciones de la función objetivo para la ejecución de todo el algoritmo ($\max_{\text{FEs}}$), número máximo de evaluaciones de la función objetivo asignadas para realizar búsqueda global por iteración (G_{FEs}) y su análogo para búsqueda local (L_{FEs}). Claramente, se puede observar que la condición de detención es verificada al final de cada generación, con el fin de terminar la ejecución del algoritmo oportunamente. No obstante, si lo que se desea es terminar la ejecución exactamente cuando se han calculado $\max_{\text{FEs}}$ evaluaciones, se pueden programar directamente a los optimizadores que se desempeñarán para realizar búsqueda global y local a fin de que verifiquen constantemente el número actual de evaluaciones.

El número máximo de iteraciones en una ejecución completa del algoritmo puede ser estimado de acuerdo a la siguiente expresión:

$$\text{it}_{\max} \leq \lceil (\max_{\text{FEs}} - \text{I}_{\text{FEs}})/(\text{G}_{\text{FEs}} + \text{L}_{\text{FEs}}) \rceil, \tag{5.1}$$

donde I_{FEs} es el número de evaluaciones empleadas durante la fase de inicialización.

Algoritmo 5.1: GL-DE

entrada: problema, $\max_{\text{FEs}}$, G_{FEs}, L_{FEs}
salida : solución

```
actual_FEs = 0
Inicializar población 1
Inicializar población 2
Actualizar: actual_FEs = actual_FEs + número de evaluaciones de la función objetivo empleadas
  durante la etapa de incialización
mientras actual_FEs < max_FEs hacer
    Evolucionar población 1 empleando evolución diferencial con un esquema especializado en
      búsqueda global tal que el proceso dispone a lo más de G_FEs evaluaciones de la función objetivo
      en esta ronda
    Actualizar: actual_FEs = actual_FEs + G_FEs
    si actual_FEs ≥ max_FEs entonces
        regresa mejor individuo de la población 1
    Migrar al mejor individuo de la población 1 hacia la población 2
    Evolucionar población 2 empleando evolución diferencial con un esquema especializado en
      búsqueda local tal que el proceso dispone a lo más de L_FEs evaluaciones de la función objetivo
      en esta ronda
    Actualizar: actual_FEs = actual_FEs + L_FEs
    si actual_FEs ≥ max_FEs entonces
        regresa mejor individuo de la población 2
    Migrar al mejor individuo de la población 2 hacia la población 1
```

5.1.1. GL-SHADE

El algoritmo GL-SHADE es una instanciación del diseño presentado en la sección pasada. En este caso, se adopta el algoritmo de evolución diferencial basado en el historial de éxito (SHADE [33], por sus siglas en inglés) como OE para realizar la fase de búsqueda

global y una variante de SHADE denominada $\text{eSHADE}_{\text{ls}}$ como OE para realizar la fase de búsqueda local. A continuación, se describe brevemente cada uno de estos OEs:

- **SHADE:** es una variante adaptativa de la evolución diferencial que incorpora una estrategia de mutación explorativa, un archivo externo que permite mayor diversidad y un mecanismo de ajuste de parámetros basado en el historial de éxito.
- **$\text{eSHADE}_{\text{ls}}$:** es una variante de SHADE que incorpora una estrategia de mutación adecuada para exploración local que además está acoplada a un buscador local basado en población. Cabe mencionar que $\text{eSHADE}_{\text{ls}}$ fue construida especialmente para realizar explotación.

Adicionalmente, se propone emplear un método de búsqueda local durante la fase de inicialización con el fin de realizar una "mejora temprana" a uno de los individuos generados más aptos. El uso de diversas técnicas de inicialización es bastante común de ver en múltiples variantes de la ED para OGGE [42]. En este caso particular, se adopta la denominada técnica de inicialización con mejora temprana (originalmente propuesta en el algoritmo de SHADE-ILS [6]). El método de búsqueda local que se usa es la popular metaheurística MTS-LS1.

A continuación, se muestra el pseudocódigo del algoritmo GL-SHADE:

Algoritmo 5.2: GL-SHADE

entrada: $f_{objetivo}$, max_{FEs}, G_{FEs}, L_{FEs}, NP_1, NP_2, H_{max_1}, H_{max_2}, w_{min}, w_{max}
salida : min f

```
// Definir componentes e inicializar sus respectivos parámetros
DE_1 ← SHADE(NP_1, H_max1, f_objetivo, max_FEs)
DE_2 ← eSHADE_ls(NP_2, H_max2, f_objetivo, max_FEs, w_min, w_max)
LS_1 ← MTS_LS1(f_objetivo, max_FEs)
actual_FEs = 0
// Inicializar: incluye inicialización de las poblaciones, las memorias M_Cr y M_F, el indice k y
   el archivo externo A
DE_1.inicializar()
DE_2.inicializar()
actual_FEs = actual_FEs + NP_1 + NP_2    // actualizar evaluaciones calculadas de la función objetivo
mejor_global ← DE_1.mejor                                  // actualizar mejor solución global
// Búsqueda local temprana (inspirado por [6])
contador_FEs = 0                                                        // reiniciar contador
LS_1.mejorar(mejor_global, L_FEs, actual_FEs, contador_FEs)          // actual_FEs cambia internamente
// Mientras no se cumpla la condición de paro...
mientras actual_FEs < max_FEs hacer
    // Búsqueda global
    DE_1.recibir(mejor_global)                        // Migrar mejor solución: desde pop 2 a pop 1
    contador_FEs = 0
    DE_1.evolucionar(G_FEs, actual_FEs, contador_FEs)          // actual_FEs cambia internamente
    mejor_global ← DE_1.mejor                              // actualizar mejor solución global
    // Búsqueda local
    DE_2.recibir(mejor_global)                       // Migrar mejor solución:: desde pop 1 a pop 2
    contador_FEs = 0
    DE_2.evolucionar(L_FEs, actual_FEs, contador_FEs)          // actual_FEs cambia internamente
    mejor_global ← DE_2.mejor                              // actualizar mejor solución global
regresa mejor_global
```

Como se puede observar, el algoritmo de GL-SHADE requiere de múltiples parámetros de entrada para su funcionamiento. A continuación, se describe brevemente cada uno de ellos a excepción de $\max_{FEs}$, G_{FEs} y L_{FEs} los cuales ya fueron presentados en la sección pasada:

- $f_{objetivo}$: la función objetivo.
- NP_1: tamaño de la población 1 (número entero > 3).
- NP_2: tamaño de la población 2 (número entero > 3).
- H_{max_1}: tamaño máximo de memoria para SHADE (número entero > 0).
- H_{max_2}: tamaño máximo de memoria para $eSHADE_{ls}$ (número entero > 0).
- w_{min}: parámetro de control para $eSHADE_{ls}$ (número real $\in [0, 1]$).
- w_{max}: parámetro de control para $eSHADE_{ls}$ (número real $\in [0, 1]$).

Es posible que la mayoría de los parámetros de control presentados resulten poco familiares. Sin embargo, por el momento basta con entender que todos ellos (a excepción de la función objetivo) corresponden a valores que deben ser definidos para el adecuado funcionamiento de los componentes. En las siguientes secciones de este capítulo se describen con sumo detalle los algoritmos de SHADE y $eSHADE_{ls}$ por lo que se espera que una vez revisadas dichas secciones quede totalmente entendido qué representa cada uno de los parámetros presentados.

La primera tarea del algoritmo de GL-SHADE es definir los componentes base que le conforman (líneas 1-3) proporcionando los parámetros adecuados. Como se puede notar, a todos los componentes se les "informa" desde un comienzo la condición de paro ($\max_{FEs}$) ya que se recomienda que sean programados de forma que puedan detener su ejecución en cuanto detecten que $actual_{FEs} \geq \max_{FEs}$. En este caso, la variable $actual_{FEs}$ es un registro para mantener el número de evaluaciones calculadas hasta el momento.

La siguiente tarea es inicializar a las poblaciones así como las correspondientes estructuras de datos requeridas para la evolución de dichas poblaciones: memorias, poblaciones externas, contador de memoria y demás. Ya creados los *demes*, se define al mejor individuo de la población 1 como el mejor global (ver línea 8) y se procede inmediatamente a efectuar el procedimiento de mejora empleando el algoritmo de MTS-LS1 (ver línea 10). Es relevante resaltar que dicho método de búsqueda local es ejecutado mientras $actual_{FEs} < \max_{FEs}$ y $contador_{FEs} < L_{FEs}$. Nótese además cómo la variable $actual_{FEs}$ es actualizada constantemente. Por tal motivo, se envía como un argumento de entrada al método de **mejorar** (ver línea 10) para que se actualice acordemente.

Posteriormente se entra a la etapa global-local. Al principio de una iteración, la población 1 recibe al mejor individuo de la población 2 (excepto en la iteración 0 donde más bien se recibe a la mejor solución recién afinada) y se integra acordemente (ver línea 12).

En este caso, el nuevo mejor individuo que se recibe desplaza al mejor individuo que se encuentra actualmente dentro de la población. Es decir, si el mejor individuo se encuentra en la posición m ($0 \leq m < NP$) entonces, el nuevo mejor se coloca en la posición m y el otro se descarta. Posteriormente, se lleva a cabo el proceso de evolución acorde con el algoritmo de SHADE (ver línea 14) el cual está programado para terminar cuando $\text{actual}_{\text{FEs}} \geq \max_{\text{FEs}}$ o $\text{contador}_{\text{FEs}} \geq \text{G}_{\text{FEs}}$. Al término de la evolución se migra al mejor individuo de la población 1 hacia la población 2 (ver líneas 15 y 16). La población 2 recibe al nuevo mejor individuo (enviado por la población 1) y lo integra a ella (ver línea 16) pero la forma de añadirlo es diferente. Ahora, en lugar de desplazar al mejor individuo que se encuentra actualmente dentro de la población, se desplaza a uno seleccionado aleatoriamente. La idea de añadirlo en una posición no fija es conservar un tipo de memoria, es decir, tener la posibilidad de acumular algunas de las mejores soluciones del pasado. Posteriormente, se lleva a cabo el proceso de evolución acorde con el algoritmo de $\text{eSHADE}_{\text{ls}}$ (ver línea 18) que de forma análoga está programado para terminar cuando $\text{actual}_{\text{FEs}} \geq \max_{\text{FEs}}$ o $\text{contador}_{\text{FEs}} \geq \text{L}_{\text{FEs}}$. Al término de la ejecución se migra al mejor individuo de la población 2 hacia la población 1 (ver líneas 19 y 12) y el ciclo se repite.

Nótese cómo durante la parte iterativa (ver líneas 11-19), la variable $\text{actual}_{\text{FEs}}$ es enviada como un argumento de entrada a los métodos **evolucionar** (ver líneas 14 y 18) para que se actualice acordemente. También obsérvese cómo la variable $\text{contador}_{\text{FEs}}$ es reiniciada previo al comienzo de un procedimiento de búsqueda (ver líneas 9, 13 y 17) y es enviada como un argumento de entrada similar a como sucede con la variable $\text{actual}_{\text{FEs}}$. Esto se debe a que la variable $\text{contador}_{\text{FEs}}$ es actualizada internamente (por los métodos de búsqueda) y su reinicio es indispensable para garantizar que un optimizador termine de ejecutarse al alcanzar el límite de evaluaciones del que dispone por iteración.

En las dos secciones siguientes se introducen y describen con detalle los algoritmos de SHADE y $\text{eSHADE}_{\text{ls}}$.

5.2. SHADE

El algoritmo de SHADE (ver [33]) es una variante de la evolución diferencial que incorpora un mecanismo especial para adaptar los parámetros de control F y Cr empleando información del historial de éxito (aquellos valores que exitosamente dieron paso a individuos más aptos), una estrategia de mutación que induce un buen nivel de diversidad en el conjunto de vectores mutantes y un archivo externo (población externa) el cual es consultado durante el procedimiento de mutación. La estrategia de mutación adoptada es efectiva para mantener un buen nivel de diversidad en el conjunto de vectores mutantes ya que se usa información de una población externa para perturbar al vector base. Otra característica notable del esquema de mutación incorporado es el uso activo de la información de los individuos más aptos de la población. Esto es conveniente ya que cualquier mejora efectuada externamente (por algún otro método) al mejor individuo puede ser

aprovechada por el motor de búsqueda interno.

Adaptación de parámetros

Cada individuo de la población ($\vec{x}_i$) está asociado a sus propios valores $\boldsymbol{Cr_i}$ y $\boldsymbol{F_i}$ tal que se genera el vector de prueba correspondiente ($\vec{u}_i$) acorde con esos valores. Al comienzo de cada generación los valores de Cr_i y F_i son configurados aleatoriamente en función de los parámetros de control adaptativos $\boldsymbol{\mu_{Cr_i}}$ y $\boldsymbol{\mu_{F_i}}$ [2] de acuerdo con la siguiente expresión:

$$\begin{aligned} Cr_i &= \text{rndN}(\mu_{Cr_i}, 0.1), \\ F_i &= \text{rndC}(\mu_{F_i}, 0.1). \end{aligned} \tag{5.2}$$

En este caso, rndN(μ, σ) y rndC(μ, σ) son valores seleccionados aleatoriamente a partir de las distribuciones **Normal** y **Cauchy** con media μ y desviación estándar σ, respectivamente. Los valores permitidos para Cr_i y F_i están acotados en los intervalos $[0, 1]$ y $(0, 1]$ respectivamente. Cuando se generan valores fuera de dichos rangos es necesario realizar un procedimiento de corrección (ver algoritmos 5.3 y 5.4).

Algoritmo 5.3: Procedimiento de corrección para F.

entrada: F_i tal que $F_i \leq 0$ o $F_i > 1$
salida : F_i tal que $F_i \in (0, 1]$

```
si F_i > 1 entonces
    F_i = 1
otro si F_i ≤ 0 entonces
    mientras F_i ≤ 0 hacer
        F_i = rndC(μ_{F_i}, 0.1)
regresa F_i
```

Algoritmo 5.4: Procedimiento de corrección para Cr.

entrada: Cr_i tal que $Cr_i < 0$ o $Cr_i > 1$
salida : Cr_i tal que $Cr_i \in [0, 1]$

```
si Cr_i > 1 entonces
    Cr_i = 1
otro si Cr_i < 0 entonces
    Cr_i = 0
regresa Cr_i
```

Los valores para μ_{Cr_i} y μ_{F_i} son seleccionados aleatoriamente de dos memorias especiales M_{Cr} y M_F. El tamaño de estas memorias (H_{max}) es un parámetro definido por el usuario que se mantiene constante durante la ejecución y sus entradas son inicializadas en 0.5.

[2] La sintaxis "μ_{Z_i}" se refiere al valor medio empleado para generar un valor del parámetro Z asociado al individuo i. Por tanto, cada individuo de la población también tiene asociados sus propios valores μ_{Cr_i} y μ_{F_i}.

El contenido de las memorias cambia a través de las generaciones (acorde con el historial de éxito): cuando en una generación G se crea un hijo ($\vec{u}_i$) estrictamente mejor que su padre ($\vec{x}_i$) entonces los valores Cr_i y F_i son almacenados temporalmente en dos historiales (S_{Cr} y S_F). Al finalizar G, las memorias M_{Cr} y M_F son actualizadas como sigue:

$$M_{Cr,k,G+1} = \begin{cases} \text{mean}_{WA}(S_{Cr}) & \text{si } S_{Cr} \neq \emptyset \\ M_{Cr,k,G} & \text{de lo contrario} \end{cases} \tag{5.3}$$

$$M_{F,k,G+1} = \begin{cases} \text{mean}_{WL}(S_F) & \text{si } S_F \neq \emptyset \\ M_{F,k,G} & \text{de lo contrario} \end{cases} \tag{5.4}$$

donde un índice k ($0 \leq k < H_{max}$) determina la entrada de la memoria que debe actualizarse. El índice k es inicializado en 0 y luego se incrementa cada vez que una entrada de la memoria es actualizada. Es importante resaltar que k cambia si y sólo si los historiales no están vacíos.

El procedimiento para calcular la **media aritmética ponderada** es como sigue:

$$\text{mean}_{WA}(S_{Cr}) = \sum_{n=0}^{|S_{Cr}|-1} w_n * S_{Cr,n}, \tag{5.5}$$

$$w_n = \frac{\Delta f_n}{\sum_{n=0}^{|S_{Cr}|-1} \Delta f_n}, \tag{5.6}$$

donde $\Delta f_n = |f(\vec{u}_{n,G}) - f(\vec{x}_{n,G})|$. Finalmente, la **media ponderada de Lehmer** se calcula como sigue:

$$\text{mean}_{WL}(S_F) = \frac{\sum_{n=0}^{|S_F|-1} w_n * S_{F,n}^2}{\sum_{n=0}^{|S_F|-1} w_n * S_{F,n}}. \tag{5.7}$$

Estrategia de mutación y archivo externo

El algoritmo de SHADE adopta la misma estrategia de mutación que el algoritmo de JADE [34] denominada *current-to-pbest/1* [3]:

$$\vec{v}_i = \vec{x}_i + F_i * (\vec{x}_{pbest} - \vec{x}_i) + F_i * (\vec{x}_{r_1} - \vec{x}_{r_2}). \tag{5.8}$$

El individuo $\vec{x}_{pbest}$ es seleccionado aleatoriamente de los $\lfloor NP * p \rfloor$ mejores individuos de la población donde $p \in (0, 1]$. En este caso, cada individuo de la población también está asociado a su propio valor p_i el cual es generado como sigue:

$$\begin{aligned} p_i &= \text{rnd}(p_{min}, 0.2) \\ \text{donde} \quad p_{min} &= \frac{2}{NP} \end{aligned} \tag{5.9}$$

[3] Los términos *current* y *target* se usan de forma indistinta.

La ventaja de esta estrategia es que enfoca la búsqueda en una región relativamente grande que está sesgada hacia direcciones de progreso prometedoras contrario a como lo haría la estrategia *rand/1* ya que esta última enfoca la búsqueda en una región relativamente pequeña que no muestra sesgos hacia ninguna dirección en especial [34].

Con el fin de promover mayor diversidad en el conjunto de vectores mutantes y, por tanto, en el conjunto de vectores hijos, se usa un archivo externo A (realmente esto es opcional) el cual está formado por vectores $\vec{x}_i$ que han sido derrotados en generaciones pasadas. El archivo A es consultado durante el procedimiento de mutación ya que el vector $\vec{x}_{r_2}$ debe ser seleccionado aleatoriamente del conjunto $P \cup A$ (unión de la población interna y externa). El tamaño de las poblaciones externa e interna son iguales, es decir, $|A| = |P| = NP$. Cuando el tamaño del archivo excede NP, elementos seleccionados al azar se eliminan hasta alcanzar el número de miembros permitido.

Pseudocódigo

A continuación se presenta el procedimiento correspondiente al algoritmo de SHADE.

Algoritmo 5.5: SHADE

entrada: NP, H_{max}, $\mathrm{f}_{\text{objetivo}}$
salida : min $\mathrm{f}_{\text{objetivo}}$

```
// método inicializar: población, memorias, índice de memorias, archivo externo, etc.
Crear una población P de forma aleatoria y uniformemente distribuida sobre Ω
M_Cr = M_F = InitMem(H_max, 0.5)                                   // inicializar memorias
A = ∅                                                              // archivo externo
k = 0                                                              // índice de memoria
// método evolucionar: realizar búsqueda
mientras la condición de paro no se cumpla hacer
    S_Cr = S_F = Δf = ∅                                            // reconfigurar historiales
    para i = 0 hasta i < NP hacer                                  // para cada individuo de la población
        r = urnd(int, 0, H_max − 1)
        μ_Cr = M_Cr,r                                              // M_Cr,r significa la entrada r de la memoria M_Cr
        μ_F = M_F,r                                                // M_F,r significa la entrada r de la memoria M_F
        Cr_i = rndN(μ_Cr, 0.1) , F_i = rndC(μ_F, 0.1)
        p = urnd(real, p_min, 0.2)
        x⃗_pbest ← seleccionar aleatoriamente con distribución uniforme uno de los mejores ⌊p ∗ NP⌋
          individuos
        Generar a u⃗_i de acuerdo con la ecuación (5.8) y empleando recombinación bin
        si f(u⃗_i) ≤ f(x⃗_i) entonces                                  // si es mejor
            P^i_nuevo ← u⃗_i                                        // hijo avanza a la siguiente generación
            si f(u⃗_i) < f(x⃗_i) entonces                            // si es estrictamente mejor
                A ← x⃗_i                                            // vector objetivo (padre) se añade al archivo externo
                S_Cr ← Cr_i , S_F ← F_i                            // añadir datos a historial
                Δf ← f(x⃗_i) − f(u⃗_i)                              // registrar diferencia
        otro
            P^i_nuevo ← x⃗_i
    P ← P_nuevo                                                    // avanzar generación
    si |A| > |P| entonces
        Eliminar miembros de A aleatoriamente hasta que |A| ≤ |P|  // mantener A
    si S_Cr ≠ ∅ y S_F ≠ ∅ entonces
        M_Cr,k = meanWA(S_Cr, Δf)
        M_F,k = meanWL(S_F, Δf)
        k = (k + 1) % H_max                                        // %: se refiere al operador módulo
regresa (x⃗ ∈ P tal que f(x⃗) ≤ f(y⃗), ∀y⃗ ∈ P \ {x⃗})                 // el mejor: x⃗_best
```

5.3. eSHADE$_{ls}$

El algoritmo de eSHADE$_{ls}$ está construido a partir del algoritmo de SHADE pero adoptando un esquema de evolución diferente e incorporando un buscador local basado en población. Este último es retomado del algoritmo **evolución diferencial mejorada basado en múltiples estrategias de mutación** (EDE [29], por sus siglas en inglés).

La variante de SHADE que se emplea corresponde al esquema $SHADE/pbest/1/exp$.

SHADE/pbest/1/exp

Este esquema corresponde al algoritmo de SHADE pero adoptando una estrategia de mutación y un tipo de recombinación diferentes.

La estrategia de mutación original de SHADE (ver ecuación 5.8) se caracteriza por generar nuevos vectores mutantes tomando en cuenta la información proporcionada por el $p\,\%$ de los mejores individuos de la población además de consultar un archivo externo conformado por individuos inferiores. Dicha estrategia es capaz de diversificar a la población para que se puedan aliviar problemas como la convergencia prematura (por ejemplo, las estrategias *current-to-best*/1 y *best*/1 sufren de este problema) [34]. Intuitivamente, a mayor diversidad, mayor capacidad explorativa y menor riesgo de quedar atrapado en un óptimo local.

De forma similar a como se modifica la estrategia *current-to-best*/1 para dar origen a la de *current-to-pbest*/1 (agregando el parámetro p que controla qué tan avariciosa se torna la búsqueda), se propone emplear la estrategia de mutación presentada en la ecuación (5.10). Esta estrategia, comparada con *best*/1, tiene menos riesgo de presentar convergencia prematura y además es adecuada para explorar la vecindad de los mejores $p\,\%$ individuos de la población. Es importante indicar que los vectores $\vec{x}_{r_1}$ y $\vec{x}_{r_2}$ son seleccionados aleatoriamente de los conjuntos P y $P \cup A$, respectivamente. Adicionalmente, con el fin de focalizar más la búsqueda, el parámetro p_i es generado aleatoriamente en el intervalo $[p_{min}, 0.1]$.

$$\vec{v}_i = \vec{x}_{pbest} + F_i * (\vec{x}_{r_1} - \vec{x}_{r_2}). \tag{5.10}$$

Finalmente, el tipo de recombinación que se incorpora es la exponencial.

EDE

El algoritmo de EDE fue propuesto como una mejora del esquema $DE/best/1/bin$. La idea es aprovechar la información producida por el mejor individuo de la población mientras se reduce el riesgo de quedar atrapado en un óptimo local (recordando que el esquema $DE/best/1/bin$ tiende a converger prematuramente). El algoritmo de EDE incorpora las dos estrategias de mutación *pbest*/1 y *current*/1 con el fin de acelerar la velocidad de convergencia y prevenir la conglomeración alrededor del mejor individuo de la población. Adicionalmente y con el objetivo de disminuir la posibilidad de quedar

atrapado en un óptimo local, se incorpora un procedimiento de perturbación sobre el mejor individuo de la población el cual es ejecutado al final de cada generación. El procedimiento correspondiente a EDE se muestra en el algoritmo 5.6.

Algoritmo 5.6: EDE

entrada: NP , Cr , F, $\mathrm{f}_{\text{objetivo}}$, max_{FEs}, r_{max}, r_{min}, w_{max}, w_{min}
salida : min $\mathrm{f}_{\text{objetivo}}$

1 $N_{FEs} = 0$
2 Inicializar una población P usando una técnica de *opposition-based learning* // consultar [29]
3 Actualizar N_{FEs} acordemente // acorde al número de evaluaciones calculadas durante la inicialización
4 **mientras** $N_{FEs} < max_{FEs}$ **hacer**
  // Ejecutar una generación de la ED
5 **para** $i = 0$ hasta $i < NP$ **hacer** // para cada individuo de la población
6 $p_{best} \leftarrow$ seleccionar aleatoriamente uno de los mejores M individuos de P // se propuso usar $M = 4$
7 Tomar dos índices $a, b \in [0, NP-1]$ aleatoriamente // $a \neq b \neq i$
8 $r1 = r_{max} - \frac{N_{FEs}}{max_{FEs}} * (r_{max} - r_{min})$ // $r_1 \in \mathbb{R}$
9 **si** $\mathrm{urnd}(\mathrm{real}, 0, 1) \leq r1$ **entonces**
10 $\vec{v_i} = \vec{x_i} + F * (\vec{x_a} - \vec{x_b})$
11 **otro**
12 $\vec{v_i} = \vec{x}_{pbest} + F * (\vec{x_a} - \vec{x_b})$
13 $\vec{u_i} \leftarrow \mathrm{recombinación\_binomial}(\vec{v_i}, \vec{x_i}, Cr)$
14 $P^i_{\text{nuevo}} \leftarrow$ el mejor entre $\vec{u_i}$ y $\vec{x_i}$
15 $N_{FEs} = N_{FEs} + 1$
16 $P \leftarrow P_{\text{nuevo}}$ // avanzar generación
  // Perturbar al mejor individuo de la población dimensión por dimensión
17 $\vec{x}_{best} \leftarrow$ el mejor individuo de P
18 **para** $j = 0$ hasta $j < D$ **hacer**
19 $\vec{\mu} = \vec{x}_{best}$ // definir a $\vec{\mu}$ como $\vec{x}_{best}$
20 $k = \mathrm{urnd}(\mathrm{int}, 0, NP-1)$, tal que $k \neq best$
21 $n = \mathrm{urnd}(\mathrm{int}, 0, D-1)$, tal que $n \neq j$
22 $r2 = w_{min} + \frac{N_{FEs}}{max_{FEs}} * (w_{max} - w_{min})$ // $r_2 \in \mathbb{R}$
23 **si** $\mathrm{urnd}(\mathrm{real}, 0, 1) \leq r2$ **entonces**
24 $\mu_j = x_{best,n} + (2 * \mathrm{urnd}(\mathrm{real}, 0, 1) - 1) * (x_{best,n} - x_{k,n})$ // $\vec{x}_k \in P$
25 **otro**
26 $\mu_j = x_{best,j} + (2 * \mathrm{urnd}(\mathrm{real}, 0, 1) - 1) * (x_{best,n} - x_{k,n})$
27 Evaluar a $\vec{\mu}$ con $\mathrm{f}_{\text{objetivo}}$ e incrementar N_{FEs}
28 Tomar al mejor entre $\{\vec{\mu}, \vec{x}_{best}\}$ para que represente a $\vec{x}_{best}$
29 **regresa** $(\vec{x} \in P$ tal que $f(\vec{x}) \leq f(\vec{y}), \forall \vec{y} \in P \setminus \{\vec{x}\})$

Los autores del algoritmo de EDE proponen emplear una técnica de inicialización denominada *opposition based learning* la cual consiste en generar $2 * NP$ puntos D-dimensionales. Los primeros NP puntos son generados aleatoriamente y con distribución uniforme mientras que los restantes son los opuestos de los primeros. Al final, se toma a los mejores NP generados (para mayores detalles revisar [29]).

Se adopta un número máximo de evaluaciones (max_{FEs}) como criterio de paro ya que este dato, además del estado de las evaluaciones que se han calculado (N_{FEs}), se usa para controlar cual de las dos estrategias de mutación tiene mayor probabilidad de ser aplicada (ver algoritmo 5.6, línea 8). Al inicio de la búsqueda, la probabilidad $r1$ se aproxima al valor de r_{max} (se usa un valor de 1.0) y cuando está cerca de su fin, dicha probabilidad se

aproxima al valor de r_{min} (se usa un valor de 0.1). Se puede observar que la probabilidad $r1$ decrece linealmente conforme avanza la búsqueda. Acorde con los valores indicados de r_{max} y r_{min} se puede concluir que al principio de la evolución, la estrategia que promueve la exploración tiene bastante mayor probabilidad de ser aplicada y, por tanto, la búsqueda se torna más explorativa que explotativa. Así mismo, conforme avanza el proceso evolutivo, la capacidad explorativa disminuye mientras que la explotativa aumenta.

El método de perturbación que se acopla al final de cada generación es prácticamente un buscador local basado en población, ya que para perturbar cada una de las dimensiones se usa información poblacional (ver algoritmo 5.6, líneas 24 y 26), que sirve para disminuir la probabilidad de quedar atrapado en un óptimo local y mejorar la velocidad de convergencia. Como se puede observar, la estrategia de perturbación a emplear es seleccionada aleatoriamente con una probabilidad que depende del número de evaluaciones calculadas y el máximo de ellas permitido (originalmente se definen $w_{min} = 0.0$ y $w_{max} = 0.2$).

Es importante notar que si se elimina la estrategia de mutación $current/1$ y por ende el parámetro $r1$, el algoritmo se convierte en un buscador local basado en población (esta es la idea que subyace al diseño de eSHADE$_{\text{ls}}$).

Pseudocódigo

A continuación se presenta el procedimiento correspondiente al algoritmo de eSHADE$_{\text{ls}}$.

Algoritmo 5.7: eSHADE$_{\text{ls}}$

entrada: max_{FEs}, NP, H_{max}, f_{objetivo}, w_{max}, w_{min}
salida : min f_{objetivo}

```
// método inicializar: población, memorias, índice de memorias, archivo externo, etc.
1 Inicializar acorde con el algoritmo de SHADE (población, memorias, archivo externo, etc.)
2 N_FEs = NP            // crear a un población requiere de NP evaluaciones de la función objetivo
// método evolucionar: realizar búsqueda
3 mientras N_FEs < max_FEs hacer
4 |   Ejecutar una generación de SHADE/pbest/1/exp
5 |   Aplicar el procedimiento de perturbación (ver algoritmo 5.6) al mejor individuo de la población
6 |   Actualizar N_FEs acorde con la cantidad de evaluaciones de la función objetivo empleadas en la
  |    iteración
7 regresa x_best
```

5.4. Manejo de restricciones de límite

Al resolver problemas de optimización global es común encontrar que las variables de decisión de las cuales depende la función objetivo están sujetas a restricciones de límite, es decir, están acotadas en un cierto intervalo. Bajo estas circunstancias, el espacio de búsqueda o región factible Ω está definido acorde a dichas restricciones (suponiendo que

sólo hay restricciones de límite).

Al emplear el algoritmo de evolución diferencial como optimizador, es posible que las variables de decisión violen sus respectivas restricciones de límite después de ser sometidas a un procedimiento de perturbación, en particular durante el procedimiento de mutación y en el caso de $\text{eSHADE}_{\text{ls}}$ también durante el procedimiento de búsqueda local basado en población (ver algoritmo 5.6, líneas 24 y 26). Por tal motivo, al término de dichos procedimientos es necesario verificar, para cada variable, que no se violen las restricciones impuestas: si la j-ésima ($0 \leq j < D$) variable del candidato a vector solución $\vec{x}_i$ está restringida en el intervalo $[lb_j, ub_j]$ (donde lb_j y ub_j son los límites inferior y superior respectivamente), entonces dicha variable viola la restricción de límite cuando $x_{i,j} < lb_j$ o $x_{i,j} > ub_j$.

En la literatura existen diversos procedimientos para corregir una variable fuera de rango, sin embargo la que se propone es como sigue [33]:

$$v_{i,j} = \begin{cases} (lb_j + x_{i,j})/2 & \text{si } v_{j,i} < lb_j, \\ (ub_j + x_{i,j})/2 & \text{si } v_{i,j} > ub_j, \end{cases} \tag{5.11}$$

donde $v_{i,j}$ se refiere a la j-ésima variable del i-ésimo vector que resulta del procedimiento de mutación donde $\vec{x}_i$ es el i-ésimo vector objetivo de la población. Análogamente al perturbar cada una de las dimensiones (ver algoritmo 5.6, líneas 24 y 26) se usa el siguiente procedimiento de corrección:

$$\mu_j = \begin{cases} (lb_j + x_{best,j})/2 & \text{si } \mu_j < lb_j. \\ (ub_j + x_{best,j})/2 & \text{si } \mu_j > ub_j. \end{cases} \tag{5.12}$$

Otro aspecto importante a considerar es la inicialización de una población donde Ω está definido acorde con restricciones de límite. En el algoritmo 5.8 se muestra el pseudocódigo para inicializar una población tomando en cuenta dichas restricciones.

Algoritmo 5.8: Inicializar población donde Ω está dado únicamente por restricciones de límite

entrada: NP, D, $\vec{x}$, $\vec{lb}$, $\vec{ub}$, $\text{f}_{\text{objetivo}}$
salida : P

```
// Como argumentos de entrada se espera el tamaño de la población, la
   dimensión del problema, un vector D-dimensional, el vector de cota
   inferior, el vector de cota superior y la función objetivo.
```

1 $P \leftarrow \emptyset$
2 **para** $i = 0$ hasta $i < NP$ **hacer**
3 | **para** $j = 0$ hasta $j < D$ **hacer**
4 | | $x_j = lb_j + (ub_j - lb_j) * \text{urnd}(\text{real}, 0, 1)$
5 | $P \leftarrow \vec{x}$
6 **regresa** P

5.5. Implementación

Normalmente, los algoritmos para optimización global a gran escala requieren un número considerable de recursos computaciones (particularmente en tiempo), sobre todo cuando el problema se resuelve como un todo (este es el caso de la nueva propuesta). No obstante, con el fin de hacer más eficiente el tiempo de ejecución de la nueva propuesta, se realizó una implementación empleando recursos paralelos, en particular unidades de procesamiento gráfico (GPUs, por sus siglas en inglés). Las secciones que son paralelizables, costosas y que operan a un nivel de paralelismo *single instruction multiple data* (SIMD, por sus siglas en inglés) son ejecutadas en el co-procesador (tarjeta gráfica o GPU) mientras que las secciones menos costosas y que no son paralelizables son ejecutadas en el procesador (CPU). Adicionalmente, la evaluación de la función objetivo es llevada a cabo por el co-procesador y, en casos particulares, por el procesador pero activando varios núcleos. Todos los detalles sobre la implementación del algoritmo de GL-SHADE pueden ser consultados en el **apéndice** A, en la página 90.

6 | Evaluación Experimental

En este capítulo se analiza el desempeño del algoritmo GL-SHADE empleando el conjunto de problemas de prueba para optimización global a gran escala más reciente denominado *CEC'13 LSGO* y realizando múltiples experimentos dirigidos a evaluar tres aspectos importantes sobre la nueva propuesta: la eficiencia [1] de la implementación, la eficiencia de la búsqueda comparado con los componentes que le conforman y la eficiencia de la búsqueda comparado con algoritmos de vanguardia.

En la sección 6.1 se presenta el conjunto de prueba adoptado mientras que en la sección 6.2 se indican los recursos de hardware y software requeridos para la elaboración de los experimentos.

En la sección 6.3 se presenta el marco experimental empleado para medir la eficiencia de la implementación de la nueva propuesta. Los resultados indican que la ganancia en velocidad de la versión paralela respecto a la versión secuencial depende del problema de prueba empleado. En el peor caso, no hay aceleración y en el mejor escenario, se tiene una ganancia de hasta 6x [2]. No obstante, en la mayoría de los casos la aceleración alcanzada es de al menos 2x.

En la sección 6.4 se presenta el marco experimental empleado para evaluar la eficiencia de búsqueda del algoritmo GL-SHADE comparado con sus componentes; los resultados indican que la nueva propuesta exhibe un mejor rendimiento general que cualquiera de ellos.

En la sección 6.5 se adopta el mismo marco experimental empleado en la sección 6.4 para comparar el rendimiento del algoritmo GL-SHADE respecto con uno de los mejores algoritmos híbridos para OGGE no basado en descomposición y que además hace uso del algoritmo de SHADE (denominado SHADE-ILS). El análisis efectuado revela que la nueva propuesta converge lento comparado con el algoritmo de SHADE-ILS resultando en un mejor rendimiento general de este último al comienzo de la búsqueda. Pero, al finalizar el proceso de optimización, el algoritmo GL-SHADE demuestra ser más competitivo en la mayoría de los problemas de prueba.

[1] Capacidad de alcanzar un objetivo en el menor tiempo posible y con el mínimo uso posible de los recursos.

[2] Una expresión para indicar una aceleración o ganancia en velocidad tal que una ejecución E_1 es 6 veces más rápida que otra ejecución E_2.

Finalmente, en la sección 6.6 se adopta el criterio empleado en la competencia de optimización global a gran escala llevada a cabo durante el congreso de cómputo evolutivo en el 2019 (*CEC LSGO competition 2019*) para comparar el desempeño de la nueva propuesta con el de múltiples algoritmos de vanguardia. Los resultados indican que el algoritmo GL-SHADE es competitivo al exhibir un rendimiento general tan bueno como el del algoritmo campeón del 2019 y obtener la mejor puntuación en los problemas traslapables.

6.1. Problemas de prueba

Actualmente existen tres conjuntos principales de problemas de prueba en el área de optimización global a gran escala:

1. *CEC'08 LSGO* [43]
2. *CEC'10 LSGO* [41]
3. *CEC'13 LSGO* [37]

El objetivo de proponer o desarrollar problemas de prueba es brindar un conjunto de referencia conveniente y flexible que permita comparar el rendimiento de varios algoritmos, en este caso propuestas diseñadas específicamente para la tarea de optimización global a gran escala.

En este capítulo se adopta el conjunto de problemas de prueba *CEC'13 LSGO*. Se decide utilizar este conjunto ya que es el más reciente y una extensión del conjunto de prueba *CEC'10 LSGO* el cual a su vez es una extensión del conjunto de prueba más antiguo (*CEC'08 LSGO*). El conjunto de prueba más reciente se distingue de sus predecesores en incorporar una gama más amplia de problemas de optimización a gran escala que se asemejan a problemas del mundo real lo que los hace más difícil de resolver (optimizar).

El conjunto de referencia *CEC'13 LSGO* está compuesto por 15 problemas de prueba; todos ellos de minimización. Los cuales están sujetos únicamente a restricciones de límite y cuentan con 1000 variables de decisión excepto por f_{13} y f_{14} los cuales son problemas traslapables donde $D = 905$. En general, estos problemas de prueba pueden ser categorizados tal como se muestra en la tabla 6.1.

El diseño, características y propiedades de todos los problemas de prueba son discutidos en el apéndice B.

6.2. Hardware y software empleados

La nueva propuesta fue desarrollada en dos versiones: paralela y secuencial. La versión paralela está implementada en el lenguaje de programación `C++/CUDA`. No obstante, tam-

Tabla 6.1: Clases de problemas incluidos en el conjunto de prueba *CEC'13 LSGO*.

Categoría	**Funciones**
Totalmente separable	f_1-f_3
Parcialmente separable	f_4-f_{11}
Traslapable	f_{12}-f_{14}
Totalmente no separable	f_{15}

bién se hace uso de `OpenMP` [3]. Por su parte la versión secuencial está implementada en el lenguaje de programación `C++`. Es importante mencionar que la versión secuencial fue desarrollada con el único propósito de llevar a cabo el experimento para medir la eficiencia de implementación de la versión paralela. Esta última es la implementación oficial y por tal motivo es la que se usa en el resto de los experimentos. El código fuente de la implementación paralela está disponible en `https://github.com/delmoral313/gl-shade`.

Todas las ejecuciones fueron llevadas a cabo en una máquina con un procesador `Intel(R) Core(TM) i7-3930K @ 3.20GHz`, con 8 `GB` de RAM y `Ubuntu 18.04` como sistema operativo. Adicionalmente, se empleó la tarjeta gráfica `GTX 680` con la versión `10.2` de `CUDA`.

6.3. Medición de la eficiencia de implementación

El experimento 1 tiene por objetivo medir la eficiencia de implementación de la versión paralela **(la estrategia y los detalles de implementación pueden ser consultados en el apéndice A)**. Para llevar a cabo el análisis se emplea la versión secuencial del algoritmo GL-SHADE como punto de referencia midiendo el factor de aceleración que resulta de dividir el tiempo de ejecución de esta última (implementación s) entre el tiempo de ejecución de la versión paralela (implementación p).

Se realizan N ejecuciones independientes (por función de prueba) las cuales están programadas para terminar al alcanzar un número máximo de evaluaciones de la función objetivo (max_{FEs}). Todas las ejecuciones son cronometradas y los respectivos tiempos de cómputo son registrados. Posteriormente, se calcula el tiempo de ejecución promedio de la implementación i empleando la función de prueba j tal como se muestra en la siguiente ecuación:

$$\bar{t}_{i,j} = \frac{\sum_{n=1}^{N} t_{n,i,j}}{N}, \tag{6.1}$$

donde $t_{n,i,j,}$ es el n-ésimo tiempo registrado de la implementación i empleando la j-ésima función de prueba. Finalmente, se calcula el factor de aceleración (ver ecuación 6.2) para cada función de prueba.

$$a_j = \frac{\bar{t}_{s,j}}{\bar{t}_{p,j}} \tag{6.2}$$

[3] `OpenMP` no es un lenguaje de programación sino mas bien una interfaz de programación de aplicaciones (API, por sus siglas en inglés) que permite multiprocesamiento en lenguajes como `C` y `C++`.

En la tabla 6.2 se muestran los tiempos de ejecución promedio de ambas implementaciones para cada problema de prueba donde $N = 25$. Adicionalmente, en las dos últimas columnas se muestran la desviación estándar y el factor de aceleración alcanzado, respectivamente. En la figura 6.1 se muestra gráficamente una comparativa, para cada función de prueba, entre el tiempo de cómputo promedio de la implementación en `C++` (barra negra) y el correspondiente tiempo de la implementación en `C++/CUDA` (barra roja). La configuración de parámetros empleada durante las ejecuciones se muestra en la tabla 6.3.

Tabla 6.2: Resultados tras realizar 25 ejecuciones del algoritmo GL-SHADE en sus dos implementaciones `C++/CUDA` y `C++` por función de prueba.

Función de prueba	Lenguaje de implementación	Tiempo de cómputo medio [s]	Desviación estándar	Aceleración
f_1	C++/CUDA	123.28	6.78	5.11
	C++	630.24	50.44	
f_2	C++/CUDA	136.12	1.41	6.38
	C++	868.04	6.48	
f_3	C++/CUDA	132.08	0.00	**6.54**
	C++	864.44	5.83	
f_4	C++/CUDA	208.04	1.73	3.72
	C++	774.56	3.46	
f_5	C++/CUDA	221.88	1.73	4.22
	C++	936.04	3.61	
f_6	C++/CUDA	223.20	1.00	4.18
	C++	933.48	3.61	
f_7	C++/CUDA	150.28	1.00	1.84
	C++	277.08	0.00	
f_8	C++/CUDA	365.56	3.16	2.47
	C++	902.48	2.83	
f_9	C++/CUDA	407.92	4.00	2.64
	C++	1075.08	5.92	
f_{10}	C++/CUDA	396.32	3.74	2.72
	C++	1077.56	5.48	
f_{11}	C++/CUDA	375.08	2.45	2.22
	C++	831.08	5.20	
f_{12}	C++/CUDA	51.80	2.24	**1.12**
	C++	57.80	1.41	
f_{13}	C++/CUDA	375.28	2.83	2.20
	C++	825.32	3.87	
f_{14}	C++/CUDA	376.76	3.87	2.19
	C++	826.76	3.74	
f_{15}	C++/CUDA	112.60	6.48	5.50
	C++	619.56	16.85	

Como se puede observar de la tabla 6.2, el mejor factor de aceleración alcanzado es de **6.54** (f_3). Es decir, la versión paralela se ejecuta 6.54 veces más rápido que la versión secuencial. No obstante, el peor factor de aceleración obtenido es de **1.12** tal como se puede observar en la gráfica correspondiente a la función de prueba f_{12} de la figura 6.1. Es fácil notar que el tiempo de cómputo depende no sólo de la implementación de las secciones paralelizables del algoritmo GL-SHADE sino también de la implementación de la función objetivo. Acorde con la definición de los problemas de prueba (ver apéndice B) hay funciones que son más costosas y demandan mayor espacio en memoria que otras (en particular, la función f_{12} no es computacionalmente costosa por lo que la ganancia en velocidad al emplear la GPU es sólo un poco más que el costo asociado a la comunicación

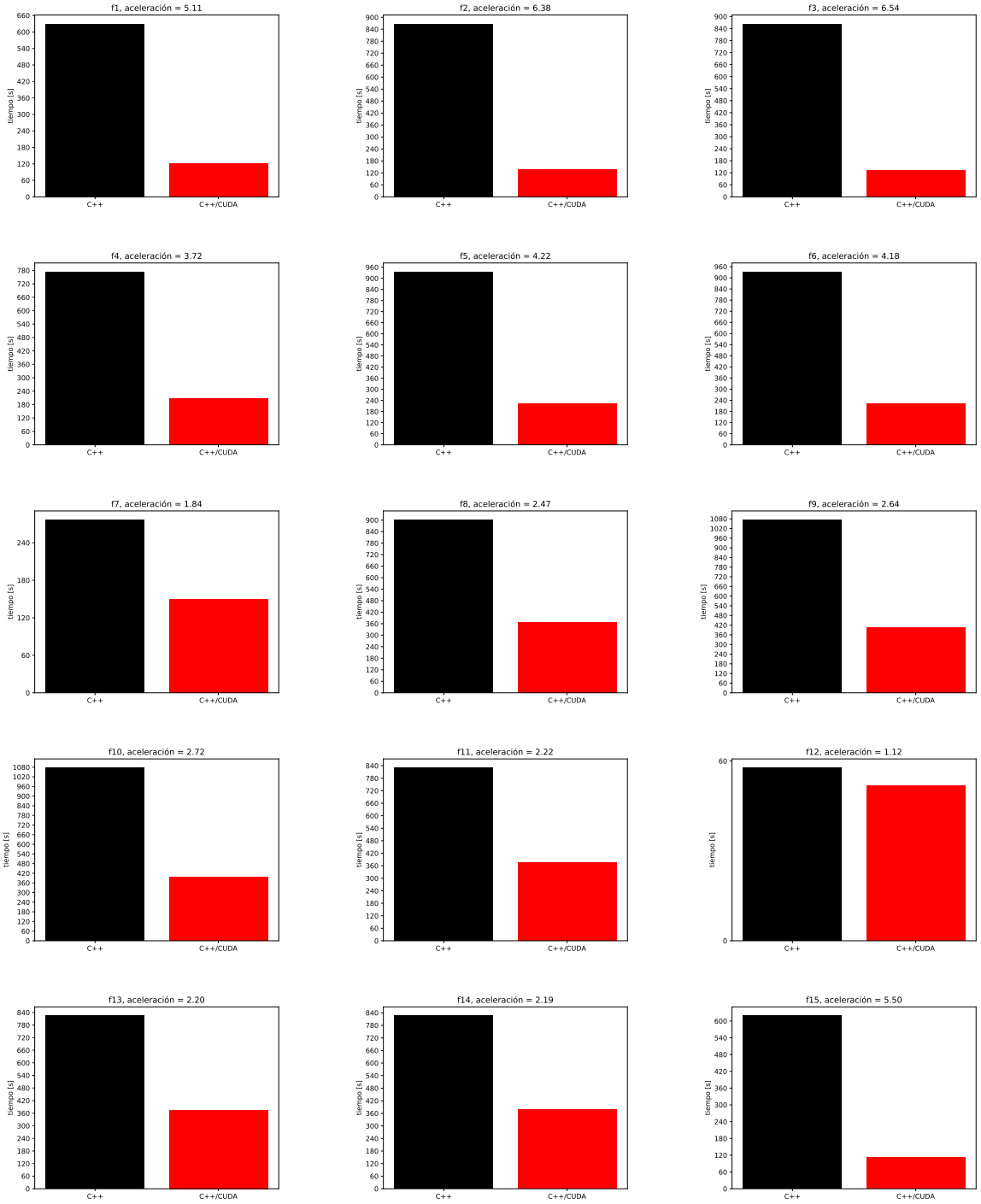

Figura 6.1: Comparativa del tiempo de ejecución promedio tras realizar 25 ejecuciones del algoritmo GL-SHADE en sus dos implementaciones `C++/CUDA` y `C++` para cada función de prueba.

entre la CPU y la GPU). Otro aspecto importante a considerar es la pérdida de velocidad que induce el uso de procedimientos de búsqueda local; generalmente no es posible paralelizarlos. En el caso del algoritmo de GL-SHADE, el método MTS-LS1 no genera un cuello de botella ya que es empleado una única vez. Sin embargo, el método de perturbación que integra la variante de la evolución diferencial eSHADE$_{\text{ls}}$ sí genera un cuello de botella. A pesar de estas dificultades es posible notar que en la mayoría de los casos la aceleración alcanzada es de al menos 2x.

Finalmente, en la tabla 6.4 y figura 6.2 se muestra el tiempo ejecución promedio que requiere la nueva propuesta (en sus dos implementaciones) para optimizar todas las funciones asociadas al conjunto de prueba adoptado. Como se puede observar, la versión paralela requiere aproximadamente **3** veces menos tiempo que la versión secuencial para optimizar todas las funciones de prueba.

Tabla 6.3: Configuración de parámetros de GL-SHADE.

Parámetro	**Valor**	**Descripción**
NB_{gpu}	32	Número de bloques solicitados al ejecutar un kernel en la GPU (**sólo para la versión paralela**).
NT_{gpu}	256	Número de hilos solicitados al ejecutar un kernel en la GPU (**sólo para la versión paralela**).
NT_{cpu}	4	Número de hilos empleados al realizar multiprocesamiento en la CPU (`OpenMP`). **Sólo para la versión paralela**.
max_{FEs}	3000000	Condición de paro. Máxima cantidad de recursos asignada para la ejecución de GL-SHADE.
G_{FEs}	25000	Recursos asignados para realizar búsqueda global por iteración
L_{FEs}	25000	Recursos asignados para realizar búsqueda local por iteración
NP_1	100	Tamaño de la población 1
NP_2	100	Tamaño de la población 2
w_{min}	0.0	Empleado por el componente eSHADE$_{\text{ls}}$ para controlar la estrategia de perturbación
w_{max}	0.2	Empleado por el componente eSHADE$_{\text{ls}}$ para controlar la estrategia de perturbación
H_{max_1}	100	Tamaño de la memoria para la población 1
H_{max_2}	100	Tamaño de la memoria para la población 2

Tabla 6.4: Tiempo de ejecución promedio requerido por GL-SHADE en sus dos implementaciones `C++/CUDA` y `C++` para optimizar todas las funciones de prueba.

Conjunto de prueba	**Lenguaje de implementación**	**Tiempo de cómputo medio [s]**	**Aceleración**
CEC'13 LSGO	`C++/CUDA` `C++`	3656.20 11499.52	3.15

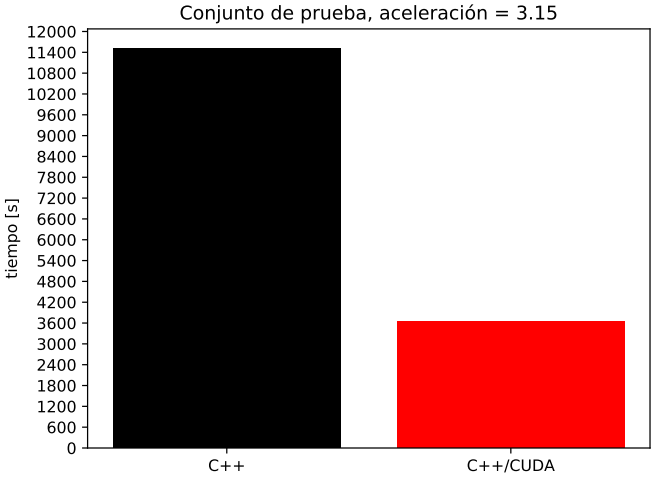

Figura 6.2: Interpretación gráfica de la tabla 6.4.

6.4. Análisis de desempeño con respecto a sus componentes

El experimento 2 tiene por objetivo medir el desempeño del algoritmo GL-SHADE y cada uno de sus componentes con el fin de investigar si la sinergia entre dichos constituyentes es realmente efectiva.

Se realizan $N = 25$ ejecuciones independientes (por problema de prueba) de la nueva propuesta y cada uno de los optimizadores que la integran, como es el caso de SHADE y eSHADE$_{\text{ls}}$, además del método MTS-LS1 el cual realmente no es parte del motor de búsqueda pero aún así se considera para el análisis.

En este caso, la implementación de todos los optimizadores es tal que están programados para reportar el mejor candidato a solución al alcanzar un cierto porcentaje de max_{FEs} (denominados puntos de control) recordando que el criterio de detención corresponde a un número límite de evaluaciones de la función objetivo. El uso de múltiples puntos de control es conveniente para analizar el estado de la mejor solución durante diferentes etapas de la búsqueda o evolución (denominados datos de convergencia).

El conjunto de puntos de control adoptado (ver [6, 44]) está conformado por 11 niveles que van desde el 4 % hasta el 100 % de max_{FEs} tal como se muestra en la tabla C.1 del apéndice C.

En la figura 6.3 se muestra una gráfica de convergencia por problema de prueba donde es posible distinguir las curvas de convergencia en escala logarítmica de los diferentes algoritmos; rojo (GL-SHADE), negro (MTS-LS1), azul (SHADE) y verde (eSHADE$_{\text{ls}}$). Como se puede observar, cada curva es construida a partir de los puntos de control, en

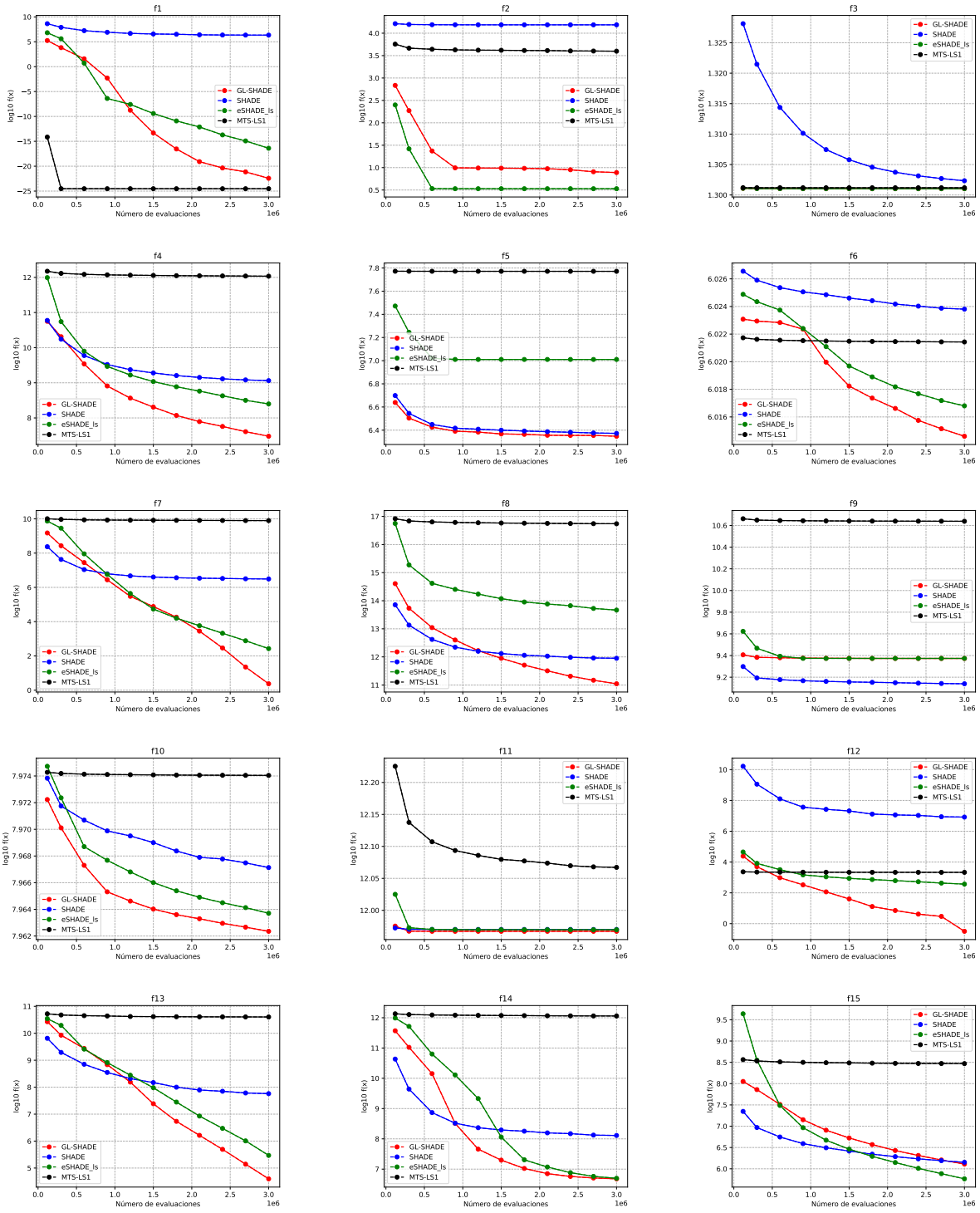

Figura 6.3: Comparativa de las curvas de convergencia de GL-SHADE y sus componentes para cada problema de prueba.

particular cada punto en una curva corresponde a la mejor solución (desempeño) promedio obtenida en diferentes etapas de la búsqueda, nótese que max_{FEs} = 3.0E+06. Los datos numéricos correspondientes al rendimiento promedio por punto de control y problema de prueba para cada algoritmo pueden ser consultados en la sección C.1, en la página 133.

La configuración de parámetros del algoritmo GL-SHADE empleada durante las ejecuciones puede ser consultada en la tabla 6.3. Por su parte, las configuraciones de parámetros de los algoritmos de SHADE y eSHADE$_{\text{ls}}$ se muestran en la tabla 6.5. Finalmente, la configuración de parámetros de MTS-LS1 es la misma adoptada en [6].

Tabla 6.5: Configuración de parámetros de SHADE y eSHADE$_{\text{ls}}$.

Parámetro	**Valor**	**Algoritmo**
NP	100	SHADE y eSHADE$_{\text{ls}}$
H_{max}	100	SHADE y eSHADE$_{\text{ls}}$
w_{min}	0.0	eSHADE$_{\text{ls}}$
w_{max}	0.2	eSHADE$_{\text{ls}}$
max_{FEs}	3000000	SHADE y eSHADE$_{\text{ls}}$

De la figura 6.3 se puede observar que el algoritmo GL-SHADE es superior al resto de optimizadores en la mayoría de los problemas de prueba (f_4-f_8, f_{10}-f_{14}) al completarse el proceso de optimización (100 % de max_{FEs}) mientras que al inicio de la búsqueda (por ejemplo, considerando el segundo punto de control que corresponde al 10 % de max_{FEs}) la nueva propuesta es superior a cualquiera de sus componentes en sólo 3 funciones de prueba (f_5, f_{10} y f_{11}). Es posible notar que conforme el proceso de búsqueda avanza, el algoritmo GL-SHADE se ve beneficiado por la estrategia de búsqueda global-local.

La información revelada por las curvas de convergencia es útil para representar y comparar el progreso de búsqueda de múltiples optimizadores. Sin embargo, no es posible distinguir si la diferencia entre ellos es realmente **significativa**.

Con el fin de investigar esto último, se lleva a cabo **la prueba de suma de rangos de Wilcoxon** entre la nueva propuesta y cada uno de sus componentes empleando un tamaño de muestra de 25 ejecuciones y un nivel de significancia estadística del 5 %; los **valores-*p*** obtenidos pueden ser consultados en el apéndice D. No obstante, en este caso solo se consideran algunos de los puntos de control (denominados los principales puntos control [4]) que corresponden al 4 %, 20 % y 100 % del máximo de evaluaciones de la función objetivo mostrados en negritas en la tabla C.1. Adicionalmente, en la sección C.2 puede ser consultado el resumen de resultados de cada optimizador considerando únicamente los principales puntos de control donde además del desempeño promedio se muestran la mediana, desviación estándar, mejor y peor solución obtenida tras 25 ejecuciones independientes.

[4] Aquellos originalmente solicitados en el reporte técnico (ver [37]) correspondiente al conjunto de prueba *CEC'13 LSGO*.

Tabla 6.6: Análisis estadístico de resultados: GL-SHADE (el algoritmo de control) vs. sus componentes usando el conjunto de funciones de prueba *CEC'13 LSGO* @1.2E+05 evaluaciones de la función objetivo.

GL-SHADE vs. sus componentes @4 % max_{FEs} Media (Sig.)				
Función	**GL-SHADE**	**MTS-LS1**	**SHADE**	**eSHADE$_{ls}$**
f_1	1.7842E+05	7.1926E-15 +	4.3294E+08 −	6.6170E+06 −
f_2	6.8434E+02	5.7012E+03 −	1.6305E+04 −	2.5071E+02 +
f_3	2.0003E+01	2.0008E+01 −	2.1288E+01 −	2.0003E+01 ≈
f_4	5.6297E+10	1.4988E+12 −	5.9577E+10 ≈	9.8731E+11 −
f_5	4.3545E+06	5.9199E+07 −	5.0000E+06 −	2.9640E+07 −
f_6	1.0546E+06	1.0513E+06 +	1.0631E+06 −	1.0590E+06 −
f_7	1.5022E+09	1.0150E+10 −	2.3728E+08 +	7.4300E+09 −
f_8	4.0515E+14	8.2773E+16 −	7.1635E+13 +	5.6340E+16 −
f_9	2.5506E+09	4.5931E+10 −	1.9875E+09 +	4.2031E+09 −
f_{10}	9.3807E+07	9.4249E+07 ≈	9.4152E+07 −	9.4347E+07 −
f_{11}	9.4476E+11	1.6800E+12 −	9.3844E+11 ≈	1.0594E+12 −
f_{12}	2.4216E+04	2.3143E+03 +	1.6504E+10 −	4.4905E+04 −
f_{13}	2.6902E+10	5.2771E+10 −	6.5152E+09 +	3.4874E+10 −
f_{14}	3.7086E+11	1.3488E+12 −	4.3136E+10 +	9.8006E+11 −
f_{15}	1.1253E+08	3.6603E+08 −	2.2297E+07 +	4.3584E+09 −
m/i/p		**3/1/11**	**6/2/7**	**1/1/13**

En la tabla 6.6 se muestra el desempeño promedio por algoritmo y problema de prueba para el primer punto de control (4 % de la búsqueda). El valor promedio va acompañado de un símbolo especial el cual indica si realmente existe o no una diferencia significativa entre los desempeños promedio sujetos a comparación. En este caso, la nueva propuesta es el algoritmo de control por lo que el símbolo "−" significa que se tiene un desempeño significativamente inferior a GL-SHADE. El símbolo "≈" indica un desempeño similar a GL-SHADE (no se encontró una diferencia estadísticamente significativa) y finalmente el símbolo "+" indica un desempeño significativamente superior a GL-SHADE. Para facilitar la lectura de la tabla, en el último renglón se añade la notación "m/i/p" la cual indica que cierto algoritmo fue significativamente mejor en "m" ocasiones, no se encontró diferencia significativa en "i" ocasiones y fue significativamente peor en "p" ocasiones.

Es posible observar que al inicio de la búsqueda la nueva propuesta exhibe un desempeño significativamente superior que los optimizadores MTS-LS1 y eSHADE$_{ls}$ en casi todos los problemas de prueba. No obstante es ligeramente superior que el algoritmo de SHADE al ser significativamente mejor en 7 problemas pero significativamente peor en 6 de ellos. Alcanzado el 20 % de la búsqueda (ver tabla 6.7), el optimizador GL-SHADE sigue siendo generalmente mejor que cualquiera de sus componentes (aunque no por mucho respecto al algoritmo de SHADE) y también se puede notar que el desempeño de la variante de la ED denominada eSHADE$_{ls}$ ha mejorado comparado con la nueva propuesta.

Finalmente, en la tabla 6.8 se observa la comparativa cuando se ha alcanzado la condición de paro (último punto de control). La nueva propuesta es significativamente mejor que el buscador local MTS-LS1 en 14 problemas de prueba, significativamente mejor que la variante adaptativa de la ED denominada SHADE en 11 problemas de prueba y significativamente mejor que el buscador local basado en población eSHADE$_{ls}$ en 7 problemas

de prueba. Por otro lado, es significativamente peor que MTS-LS1 en 0 problemas de prueba, significativamente peor que SHADE en 1 problema de prueba y significativamente peor que $\text{eSHADE}_{\text{ls}}$ en 1 problema de prueba. Estos resultados indican que el algoritmo GL-SHADE exhibe un mejor rendimiento general que SHADE y $\text{eSHADE}_{\text{ls}}$ considerados por separado. Por tanto, la sinergia entre los esquemas evolutivos que se proponen para construir el motor de búsqueda de GL-SHADE es efectiva.

Tabla 6.7: Análisis estadístico de resultados: GL-SHADE (el algoritmo de control) vs. sus componentes usando el conjunto de funciones de prueba *CEC'13 LSGO* @6.0E+05 evaluaciones de la función objetivo.

GL-SHADE vs. sus componentes @20% max_{FEs} Media (Sig.)				
Función	**GL-SHADE**	**MTS-LS1**	**SHADE**	**$\text{eSHADE}_{\text{ls}}$**
f_1	3.8196E+01	3.0182E-25 +	1.7272E+07 −	5.0671E+00 +
f_2	2.3512E+01	4.3831E+03 −	1.5473E+04 −	3.3988E+00 +
f_3	2.0000E+01	2.0007E+01 −	2.0624E+01 −	2.0000E+01 +
f_4	3.4434E+09	1.2212E+12 −	5.9757E+09 −	7.9420E+09 −
f_5	2.6669E+06	5.9126E+07 −	2.8134E+06 ≈	1.0498E+07 −
f_6	1.0540E+06	1.0509E+06 +	1.0601E+06 −	1.0562E+06 −
f_7	2.8045E+07	8.7338E+09 −	1.0935E+07 +	9.0724E+07 −
f_8	1.1049E+13	6.3760E+16 −	4.1950E+12 +	4.1664E+14 −
f_9	2.3967E+09	4.4213E+10 −	1.5042E+09 +	2.4750E+09 ≈
f_{10}	9.2747E+07	9.4219E+07 −	9.3474E+07 −	9.3047E+07 ≈
f_{11}	9.2732E+11	1.2801E+12 −	9.3355E+11 ≈	9.3346E+11 ≈
f_{12}	9.6040E+02	2.2080E+03 −	1.2691E+08 −	3.2014E+03 −
f_{13}	2.7300E+09	4.5008E+10 −	7.0836E+08 +	2.5832E+09 ≈
f_{14}	1.4210E+10	1.2339E+12 −	7.3684E+08 +	6.3330E+10 −
f_{15}	3.2416E+07	3.2379E+08 −	5.5794E+06 +	3.0700E+07 ≈
m/i/p		**2/0/13**	**6/2/7**	**3/5/7**

Tabla 6.8: Análisis estadístico de resultados: GL-SHADE (el algoritmo de control) vs. sus componentes usando el conjunto de funciones de prueba *CEC'13 LSGO* @3.0E+06 evaluaciones de la función objetivo.

GL-SHADE vs. sus componentes @max_{FEs} Media (Sig.)				
Función	**GL-SHADE**	**MTS-LS1**	**SHADE**	**$\text{eSHADE}_{\text{ls}}$**
f_1	3.7379E-23	3.0182E-25 ≈	2.1845E+06 −	4.0663E-17 −
f_2	7.7610E+00	3.9680E+03 −	1.5379E+04 −	3.3829E+00 ≈
f_3	2.0000E+01	2.0007E+01 −	2.0060E+01 −	2.0000E+01 ≈
f_4	3.0118E+07	1.0820E+12 −	1.1522E+09 −	2.5036E+08 −
f_5	2.2264E+06	5.9088E+07 −	2.3572E+06 ≈	1.0247E+07 −
f_6	1.0342E+06	1.0506E+06 −	1.0564E+06 −	1.0395E+06 ≈
f_7	2.3673E+00	7.9165E+09 −	3.1042E+06 −	2.6734E+02 −
f_8	1.1064E+11	5.4924E+16 −	8.9442E+11 −	4.6405E+13 −
f_9	2.3615E+09	4.3535E+10 −	1.3792E+09 +	2.3639E+09 ≈
f_{10}	9.1697E+07	9.4198E+07 −	9.2711E+07 −	9.1982E+07 ≈
f_{11}	9.2729E+11	1.1674E+12 −	9.3339E+11 ≈	9.3258E+11 ≈
f_{12}	3.1896E-01	2.1429E+03 −	8.2921E+06 −	3.6988E+02 −
f_{13}	3.9831E+04	4.0259E+10 −	5.7585E+07 −	2.9525E+05 −
f_{14}	4.7868E+06	1.1431E+12 −	1.2944E+08 −	4.9800E+06 ≈
f_{15}	1.2919E+06	2.9521E+08 −	1.4229E+06 ≈	5.8184E+05 +
m/i/p		**0/1/14**	**1/3/11**	**1/7/7**

6.5. Análisis de desempeño respecto con SHADE-ILS

El experimento 3 tiene por objetivo comparar el desempeño de la nueva propuesta con uno de los mejores algoritmos híbridos para OGGE de la literatura [39], denominado SHADE-ILS (algoritmo introducido en el capítulo 4, sección 4.5). El algoritmo SHADE-ILS, al igual que GL-SHADE, hace uso de SHADE como componente explorativo. No obstante, difieren fuertemente en dos aspectos importantes:

1. El algoritmo SHADE-ILS hace uso de dos métodos para realizar búsqueda local (explotación): MTS-LS1 y L-BFGS-B. El primero de ellos es un método de búsqueda local no basado en población y no basado en gradiente mientras que el segundo es un método basado en gradiente, es decir, usa información de la derivada (en realidad, de una aproximación a ella) para guiar la búsqueda. En el caso del algoritmo GL-SHADE, se emplea una variante de la evolución diferencial ($\text{eSHADE}_{\text{ls}}$) como único componente explotativo.

2. El algoritmo SHADE-ILS incorpora un mecanismo especial de reinicio el cual es activado al detectarse un estancamiento (atrapado en un óptimo local) mientras que la nueva propuesta no incorpora nada similar. En el caso del algoritmo GL-SHADE se trata el problema del estancamiento por medio del componente $\text{eSHADE}_{\text{ls}}$ el cual integra un mecanismo de perturbación al final de cada generación que está pensado justamente para aumentar la probabilidad de escapar de un óptimo local en caso de caer en uno.

Para analizar el rendimiento de los optimizadores se adopta el mismo marco experimental presentado en la sección pasada. Se realizan 25 ejecuciones independientes por problema de prueba reportando el mejor candidato a solución acorde con el conjunto de puntos de control previamente presentados (ver tabla C.1) con el fin de generar los datos de convergencia. Así mismo, los datos numéricos asociados a las curvas de convergencia de los algoritmos SHADE-ILS y GL-SHADE pueden ser consultados en el apéndice C, sección C.1.

Posteriormente, para verificar si existe una diferencia estadísticamente significativa entre el desempeño promedio de los optimizadores sujetos a comparación se lleva a cabo la prueba de suma de rangos de Wilcoxon empleando un tamaño de muestra de 25 ejecuciones y un nivel de significancia estadística del 5 %; los valores-p obtenidos pueden ser consultados en el apéndice D. De igual forma, en la sección C.2 puede ser consultado el resumen de resultados de SHADE-ILS considerando únicamente los principales puntos de control (4 %, 20 % y 100 %) donde además del desempeño promedio se muestran la mediana, desviación estándar, mejor y peor solución obtenidas tras 25 ejecuciones independientes.

La configuración de parámetros del algoritmo GL-SHADE empleada durante las ejecuciones es la que se muestra en la tabla 6.3 y la configuración de parámetros del algoritmo

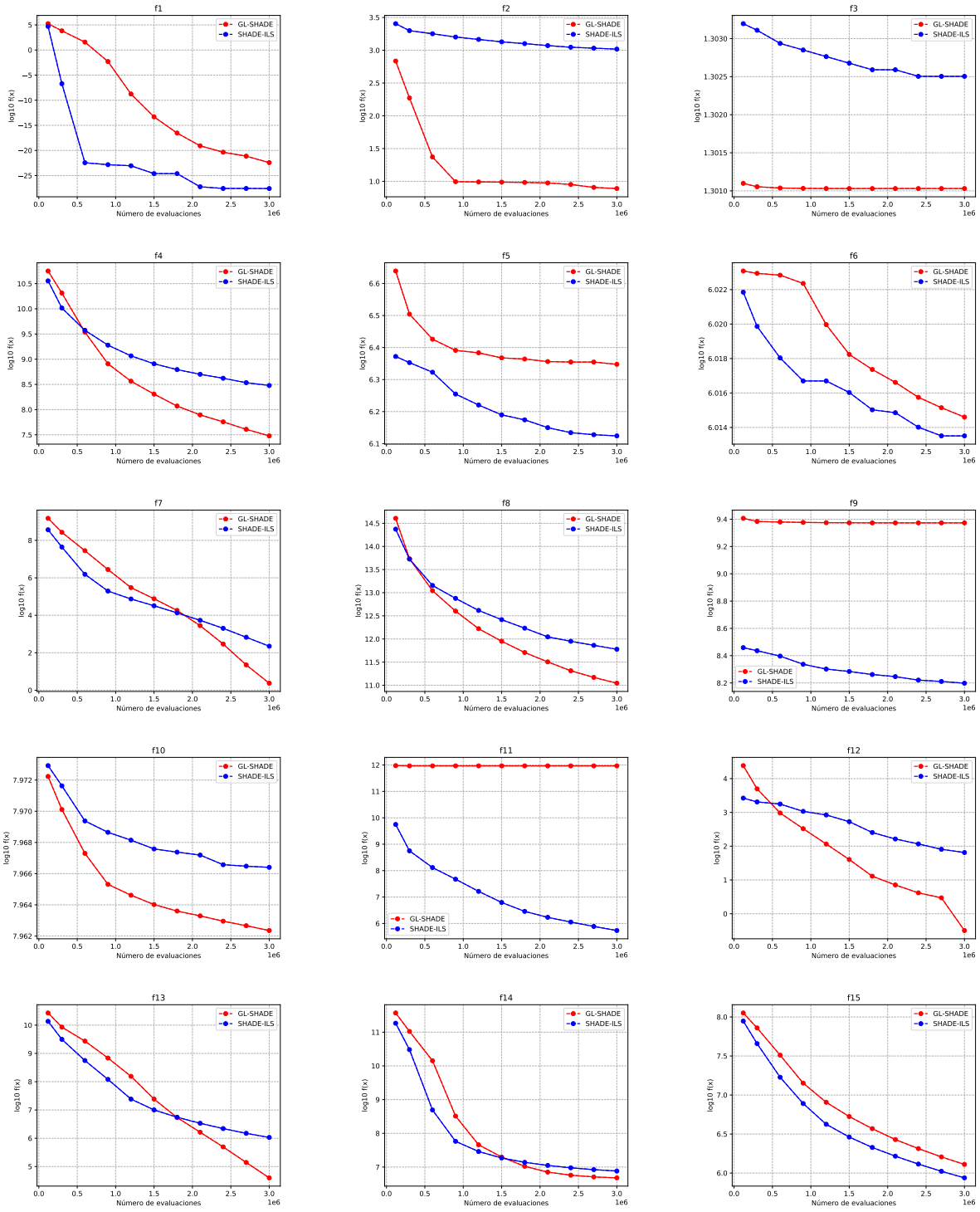

Figura 6.4: Comparativa de las curvas de convergencia de GL-SHADE y SHADE-ILS para cada problema de prueba.

SHADE-ILS es la que se indica en [6]. El código fuente correspondiente al algoritmo SHADE-ILS puede ser obtenido de: `https://github.com/dmolina/shadeils`.

En la figura 6.4 se muestran las gráficas de convergencia de SHADE-ILS (azul) y GL-SHADE (rojo) mientras que en las tablas 6.9, 6.10 y 6.11 se muestran los resúmenes del análisis de desempeño entre dichos optimizadores para los puntos de control 4 %, 20 % y 100 % de max_{FEs}, respectivamente.

En este caso, SHADE-ILS es el algoritmo de control por lo que el símbolo "+" indica que GL-SHADE presenta un desempeño significativamente superior que SHADE-ILS. El símbolo "≈" indica un desempeño similar y el símbolo "−" indica un desempeño significativamente inferior que SHADE-ILS.

Como se puede observar de las tablas 6.9 y 6.10, al inicio de la búsqueda, el algoritmo de SHADE-ILS demuestra ser más competitivo que la nueva propuesta en la mayoría de los problemas de prueba. No obstante, al finalizar el proceso de optimización (ver tabla 6.11) la nueva propuesta es interesantemente mejor que SHADE-ILS en 9 problemas de prueba mientras que este último lo es en sólo 4 de ellos. En particular, GL-SHADE muestra ser significativamente mejor que SHADE-ILS en los problemas traslapables.

Estos resultados indican que la nueva propuesta converge más lento que SHADE-ILS. Sin embargo, en la mayoría de los casos es una convergencia sostenida lo cual permite que una vez completado el proceso de optimización, GL-SHADE supere a SHADE-ILS en la mayoría de los problemas de prueba.

En realidad no es sorprendente que la nueva propuesta converja lento comparado con SHADE-ILS ya que este último integra dos métodos de búsqueda puramente locales los cuales consideran un único punto dentro de la región factible e intentan afinarlo iterativamente guiando la búsqueda acorde con la información del gradiente (en el caso del algoritmo L-BFGS-B) o las aparentemente mejores decisiones que pueden ser tomadas en un cierto momento (en el caso de MTS-LS1). Los métodos de búsqueda de este tipo son más poderosos (sobre todo MTS-LS1) que el buscador local basado en población $\text{eSHADE}_{\text{ls}}$ pero también son más propensos a quedar atrapados en un óptimo local ya que $\text{eSHADE}_{\text{ls}}$ usa información poblacional para mejorar el punto sujeto a explotación mientras que MTS-LS1 y L-BFGS-B no. Como consecuencia, se tiene una tasa de convergencia más lenta pero con mayor probabilidad de escapar de un óptimo local.

Tabla 6.9: Análisis estadístico de resultados: SHADE-ILS (el algoritmo de control) vs. GL-SHADE usando el conjunto de funciones de prueba *CEC'13 LSGO* @1.2E+05 evaluaciones de la función objetivo.

SHADE-ILS vs. GL-SHADE @4% max_{FEs} Media ±DevSt.			
Función	**SHADE-ILS**	**GL-SHADE**	**Sig.**
f_1	**5.1296e+04 ±2.7921e+04**	1.7842e+05 ±7.2299e+04	−
f_2	2.5452e+03 ±1.4142e+02	**6.8434e+02 ±3.0459e+01**	+
f_3	2.0100e+01 ±0.0000e+00	**2.0003e+01 ±3.5845e-04**	+
f_4	**3.5864e+10 ±1.5187e+10**	5.6297e+10 ±2.0058e+10	−
f_5	**2.3556e+06 ±3.6658e+05**	4.3545e+06 ±5.0855e+05	−
f_6	1.0516e+06 ±6.2450e+03	1.0546e+06 ±3.6941e+03	≈
f_7	**3.6856e+08 ±9.7432e+07**	1.5022e+09 ±5.8094e+08	−
f_8	**2.3631e+14 ±1.4782e+14**	4.0515e+14 ±2.4962e+14	−
f_9	**2.8760e+08 ±3.7500e+07**	2.5506e+09 ±6.6079e+08	−
f_{10}	9.3956e+07 ±6.4036e+05	9.3807e+07 ±4.7359e+05	≈
f_{11}	**5.5884e+09 ±3.8645e+09**	9.4476e+11 ±1.8168e+10	−
f_{12}	**2.6464e+03 ±3.0895e+02**	2.4216e+04 ±3.1165e+03	−
f_{13}	**1.3623e+10 ±6.7078e+09**	2.6902e+10 ±6.9505e+09	−
f_{14}	**1.8416e+11 ±9.0309e+10**	3.7086e+11 ±1.7758e+11	−
f_{15}	**8.8616e+07 ±2.4589e+07**	1.1253e+08 ±1.2393e+07	−
m/i/p	**11/2/2**	**2/2/11**	

Tabla 6.10: Análisis estadístico de resultados: SHADE-ILS (el algoritmo de control) vs. GL-SHADE usando el conjunto de funciones de prueba *CEC'13 LSGO* @6.0E+05 evaluaciones de la función objetivo.

SHADE-ILS vs. GL-SHADE @20% max_{FEs} Media ±DevSt.			
Función	**SHADE-ILS**	**GL-SHADE**	**Sig.**
f_1	**3.5512e-23 ±6.3958e-23**	3.8196e+01 ±4.8843e+01	−
f_2	1.7864e+03 ±1.9213e+02	**2.3512e+01 ±8.5986e+00**	+
f_3	2.0088e+01 ±3.3166e-02	**2.0000e+01 ±3.1765e-05**	+
f_4	3.7448e+09 ±1.0749e+09	3.4434e+09 ±2.5806e+09	≈
f_5	**2.1032e+06 ±3.8091e+05**	2.6669e+06 ±3.2901e+05	−
f_6	**1.0424e+06 ±7.2342e+03**	1.0540e+06 ±3.6766e+03	−
f_7	**1.5421e+06 ±5.8583e+05**	2.8045e+07 ±1.3186e+07	−
f_8	1.4339e+13 ±7.1935e+12	1.1049e+13 ±7.1567e+12	≈
f_9	**2.4932e+08 ±3.7174e+07**	2.3967e+09 ±6.6920e+08	−
f_{10}	9.3192e+07 ±6.3831e+05	**9.2747e+07 ±5.2362e+05**	+
f_{11}	**1.3048e+08 ±3.4470e+07**	9.2732e+11 ±1.0629e+10	−
f_{12}	1.7688e+03 ±2.7114e+02	**9.6040e+02 ±4.1194e+02**	+
f_{13}	**5.6046e+08 ±3.1299e+08**	2.7300e+09 ±1.0123e+09	−
f_{14}	**4.9090e+08 ±6.2008e+08**	1.4210e+10 ±1.5361e+10	−
f_{15}	**1.6893e+07 ±1.3683e+07**	3.2416e+07	−
m/i/p	**9/2/4**	**4/2/9**	

Tabla 6.11: Análisis estadístico de resultados: SHADE-ILS (el algoritmo de control) vs. GL-SHADE usando el conjunto de funciones de prueba *CEC'13 LSGO* @3.0E+06 evaluaciones de la función objetivo.

SHADE-ILS vs. GL-SHADE @max_{FEs} Media ±DevSt.			
Función	**SHADE-ILS**	**GL-SHADE**	**Sig.**
f_1	**2.5558e-28 ±5.3619e-28**	3.7379e-23 ±1.1330e-22	−
f_2	1.0415e+03 ±1.0341e+02	**7.7610e+00 ±1.0094e+01**	+
f_3	2.0068e+01 ±4.7610e-02	**2.0000e+01 ±0.0000e+00**	+
f_4	3.0128e+08 ±1.0458e+08	**3.0118e+07 ±1.7759e+07**	+
f_5	**1.3310e+06 ±2.2657e+05**	2.2264e+06 ±3.6616e+05	−
f_6	1.0316e+06 ±9.8658e+03	1.0342e+06 ±1.0735e+04	≈
f_7	2.2356e+02 ±2.5286e+02	**2.3673e+00 ±2.7597e+00**	+
f_8	5.9937e+11 ±5.4287e+11	**1.1064e+11 ±1.4976e+11**	+
f_9	**1.5780e+08 ±1.4888e+07**	2.3615e+09 ±6.6534e+08	−
f_{10}	9.2556e+07 ±4.5100e+05	**9.1697e+07 ±4.6953e+05**	+
f_{11}	**5.3888e+05 ±2.2303e+05**	9.2729e+11 ±1.0580e+10	−
f_{12}	6.4886e+01 ±2.2987e+02	**3.1896e-01 ±1.1038e+00**	+
f_{13}	1.0706e+06 ±8.6269e+05	**3.9831e+04 ±2.9867e+04**	+
f_{14}	7.6280e+06 ±1.2756e+06	**4.7868e+06 ±2.1480e+05**	+
f_{15}	8.6832e+05 ±6.3444e+05	1.2919e+06 ±1.1058e+06	≈
m/i/p	**4/2/9**	**9/2/4**	

6.6. Comparativa con algoritmos del estado del arte

Hasta el momento se ha mostrado que GL-SHADE es una propuesta construida a partir de combinar efectivamente diferentes esquemas evolutivos y también que es competitivo e incluso superior en la mayoría de los problemas de prueba que una de las mejores propuestas híbridas presentes en la literatura (SHADE-ILS) para optimización global a gran escala. Con el fin de extender el análisis de desempeño de GL-SHADE, se elabora un estudio donde se hace una comparativa entre su rendimiento y el de múltiples algoritmos del estado del arte para OGGE.

El conjunto de algoritmos de vanguardia que se consideran para este estudio (ver tabla 6.12) está constituido por propuestas recientes y que han tenido un desempeño sobresaliente en las competencias recientes de OGGE llevadas a cabo durante varias ediciones del *IEEE Congress on Evolutionary Computation*. Una breve descripción de los algoritmos mostrados en la tabla 6.12 (a excepción de CC-RDG3) se proporciona en el capítulo 4, sección 4.5. Respecto a la propuesta CC-RDG3, es un algoritmo que emplea el marco de coevolución cooperativa (CC) e incorpora una técnica de descomposición denominada agrupación diferencial recursiva en su versión 3 (RDG3, por sus siglas en inglés). Aquí se hace referencia a la **versión 3** ya que RDG3 nace como una modificación de versiones previas (RDG2 y RDG). Para los lectores interesados en una descripción detallada de la técnica RDG3 se sugiere revisar [23].

Tabla 6.12: Conjunto de algoritmos de referencia que han tenido un desempeño sobresaliente en las competencias recientes de OGGE llevadas a cabo durante varias ediciones del *IEEE Congress on Evolutionary Computation*.

Algoritmo	**Año de participación**	**Descripción**
MOS	2013	Híbrido. Ganador desde el 2013 hasta 2017 [38].
MLSHADE-SPA	2018	Coevolutivo cooperativo e híbrido. Segundo lugar en la competencia del 2018 [38].
SHADE-ILS	2018	Híbrido. Primer lugar en la competencia del 2018 [38].
CC-RDG3	2019	Coevolutivo cooperativo. Primer lugar en la competencia del 2019 [44].

La comparativa de rendimientos es llevada a cabo empleando una herramienta *online* avanzada para contrastar el desempeño de diferentes metaheurísticas (propuesta en [45]). Esta herramienta es empleada por los organizadores de las competencias en OGGE previamente mencionadas. En particular, fue empleada en la reciente *IEEE CEC LSGO competition 2019* [44]. El sitio *web* de dicha herramienta (denominada TACO, por sus siglas en inglés) es `https://tacolab.org/` y desde ahi es posible generar reportes especiales para contrastar el rendimiento de diversos algoritmos tomando en cuenta un conjunto estándar de prueba. La herramienta TACO tiene acceso a una base de datos donde se cuenta con registros de desempeño de múltiples algoritmos que han participado en las competencias de OGGE.

Para generar los reportes de rendimiento se requiere cargar un archivo con información sobre el desempeño de GL-SHADE el cual contiene la mejor solución promedio por problema de prueba para los puntos de control 4 %, 20 % y 100 % de max_{FEs} donde $max_{FEs} =$ 3.0E+06. En este caso, se emplea el conjunto estándar de prueba *CEC'13 LSGO* y se usa el conjunto de algoritmos de referencia mostrado en la tabla 6.12. El reporte de rendimiento generado se muestra en la figura 6.5 donde es posible distinguir gráficamente el nivel de desempeño (puntaje) de los "competidores". Como se puede observar, cada color corresponde a una clase específica de problemas y el nivel de la barra indica el puntaje alcanzado en esa clase en particular. La suma de los puntajes particulares resulta en el puntaje final.

El criterio adoptado por TACO (y los organizadores de las competencias) para la asignación de puntajes es análogo a como se maneja en las competencias de autos de fórmula uno (FOS, por sus siglas en inglés): el proceso de optimizar una función es análogo a una carrera por lo que al competidor (algoritmo) que queda en primer lugar (mejor desempeño promedio) se le otorgan 25 puntos, al segundo 18 puntos, al tercero 15 puntos, al cuarto 12 puntos y al quinto 10 puntos.

En la tabla 6.13 se muestran los puntajes obtenidos por clase de problemas además del puntaje final acumulado para cada algoritmo considerando únicamente el último punto

Tabla 6.13: Puntaje final y por clase de problemas de acuerdo con el criterio FOS.

Puntaje @3.0E+06 evaluaciones de acuerdo con FOS					
	CC-RDG3	**GL-SHADE**	**MLSHADE-ILS**	**MOS**	**SHADE-ILS**
Clase separable (f_1-f_3)	30	55	55	**58**	42
Clase parcialmente separable con un subcomponente separable (f_4-f_7)	**80**	58	63	52	67
Clase parcialmente separable con ningún subcomponente separable (f_8-f_{11})	**90**	50	68	54	58
Clase traslapable (f_{12}-f_{14})	38	**75**	40	36	51
Clase no separable (f_{15})	15	18	10	12	**25**
Total					
Puntaje total	253	**256**	236	212	243
Posición	#2	**#1**	#4	#5	#3

de control (al cumplirse la condición de paro).

Estos resultados indican que el algoritmo de GL-SHADE es inferior que la mayoría de los competidores al comienzo del proceso de optimización (ver primeras dos gráficas de la figura 6.5). Tal como se indicó en la sección pasada, la nueva propuesta se caracteriza por tener un inicio lento comparado con otros algoritmos lo cual no es algo necesariamente malo ya que GL-SHADE incorpora una variante de la evolución diferencial como buscador local. A diferencia de un buscador local no poblacional, esta variante adopta una estrategia de búsqueda bastante menos avariciosa (idealmente ayuda a prevenir la convergencia prematura).

Cuando el proceso de optimización llega al 100 % (ver la última gráfica de la figura 6.5 y la tabla 6.13) es posible observar que GL-SHADE pasó de ser el peor competidor a convertirse en el mejor. Todo indica que GL-SHADE no converge prematuramente en la mayoría de los casos y aunque converge lentamente, lo hace de forma sostenida. También es posible notar que cada algoritmo se desempeña mejor en una clase específica de problemas: MOS es el mejor para problemas separables, CC-RDG3 para problemas parcialmente separables, GL-SHADE para problemas traslapables y SHADE-ILS para problemas no separables. Por lo tanto, acorde con los estándares de desempeño adoptados en la competencia *IEEE CEC LSGO competition 2019*, la nueva propuesta es un algoritmo competitivo y uno de los mejores para atacar problemas traslapables.

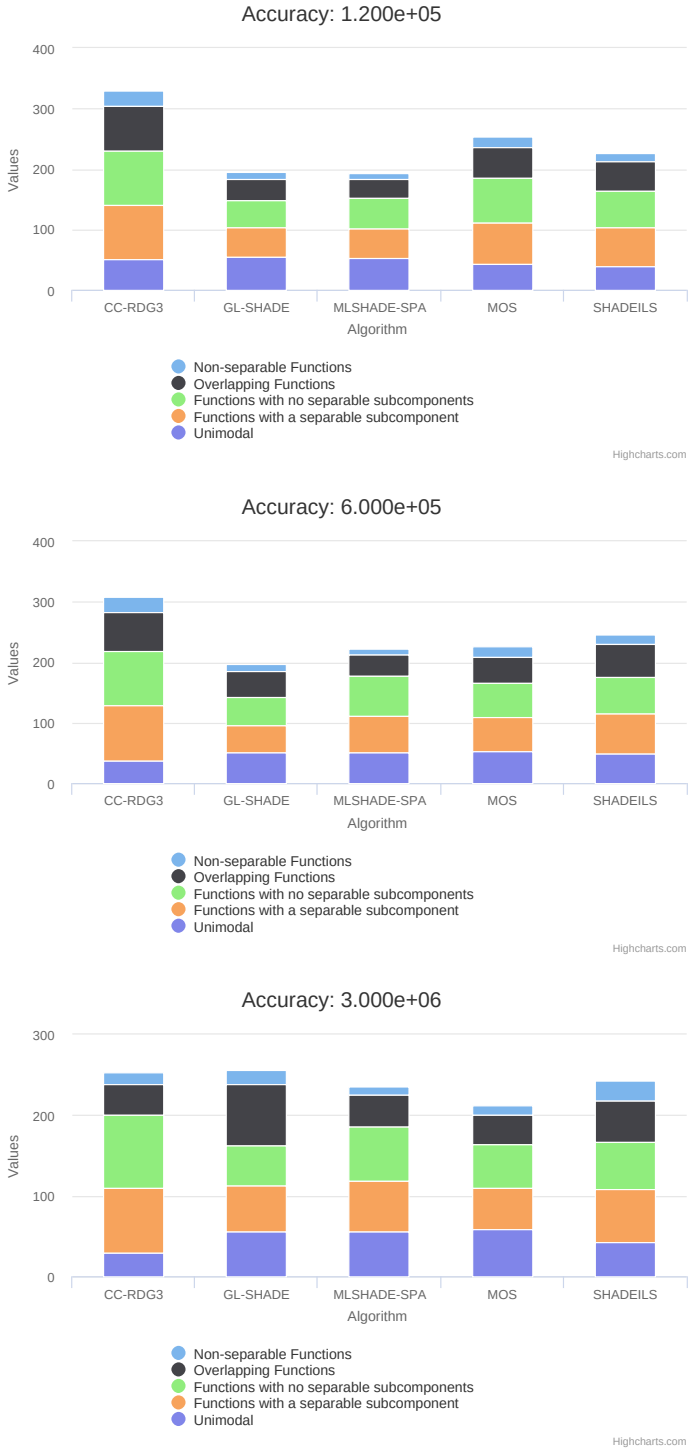

Figura 6.5: GL-SHADE vs. algoritmos de vanguardia empleando el conjunto de prueba *CEC'13 LSGO* y adoptando el criterio FOS.

7 | Conclusiones y trabajo futuro

En este trabajo se propuso un nuevo esquema basado en el algoritmo de evolución diferencial para optimización global con 1000 variables. Particularmente, se basa en una de sus variantes adaptativas denominada evolución diferencial con adaptación de parámetros basado en el historial de éxito (SHADE, por sus siglas en inglés). La nueva propuesta, denominada GL-SHADE, es un algoritmo no basado en descomposición cuyo motor de búsqueda está construido a partir de combinar dos esquemas complementarios de evolución.

El algoritmo GL-SHADE divide el proceso de búsqueda en tres pasos: (1) inicialización, (2) búsqueda global y (3) búsqueda local. El procedimiento de inicialización se distingue por realizar una búsqueda local temprana (se emplea el método MTS-LS1) posterior a generar un conjunto de individuos con el fin de mejorar al candidato a solución más apto creado. Los pasos de búsqueda global y local se repiten iterativamente hasta cumplirse la condición de paro. Esto lo realiza un motor de búsqueda global-local o fase evolutiva combinada donde SHADE es empleado como motor de búsqueda global y, a diferencia de la mayoría de los algoritmos híbridos propuestos, se incorpora una nueva variante de la evolución diferencial como motor de búsqueda local. Esta última fue construida a partir de las variantes SHADE y EDE denominada $\text{eSHADE}_{\text{ls}}$.

Con el fin de obtener tiempos de ejecución más cortos se propuso implementar el algoritmo GL-SHADE usando una tarjeta de aceleración gráfica (GPU).

Se realizaron múltiples experimentos adoptando un conjunto de funciones de prueba (*CEC'13 LSGO*) estándar en el área de optimización global a gran escala dirigidos a evaluar tres aspectos importantes de GL-SHADE: (1) eficiencia de la implementación, (2) eficiencia de la búsqueda comparado con sus componentes y (3) eficiencia de la búsqueda comparado con algoritmos del estado del arte.

Para medir la eficiencia de la implementación, se realizó un comparativo entre el tiempo de ejecución promedio de las implementaciones paralela y secuencial de GL-SHADE por problema de prueba. De los resultados reportados, se observó que, en el mejor de los casos, la versión paralela obtiene una aceleración de 6x y en el peor de los casos no hay aceleración. Se pudo notar que el factor de aceleración depende del costo computacional asociado al problema de prueba adoptado como función objetivo. Hay casos en los que la ganancia en velocidad es solo un poco mayor que el costo asociado a la comunicación

entre la CPU y GPU. También se indicó la pérdida de aceleración que inducen los métodos de búsqueda local; generalmente no son paralelizables y, por tanto, generan cuellos de botella. A pesar de estas dificultades, se observó que en la mayoría de los problemas de prueba la implementación paralela es, en promedio, dos veces más rápida que la implementación secuencial. Finalmente, se mostró que la versión paralela puede optimizar todas las funciones de prueba tres veces más rápido que su análoga secuencial.

Para analizar la eficiencia de búsqueda de GL-SHADE respecto a los componentes que le conforman (MTS-LS1, SHADE y $\text{eSHADE}_{\text{ls}}$) se comparó el rendimiento promedio (mejor solución promedio) considerando múltiples puntos de control (etapas de búsqueda). Los resultados obtenidos indicaron que la nueva propuesta exhibe un mejor rendimiento promedio (estadísticamente validado) que cualquiera de sus componentes por separado en la mayoría de los problemas de prueba; no sólo al comienzo del proceso de optimización (búsqueda) sino también al término de éste. Estos resultados muestran que GL-SHADE aprovecha correctamente el uso de los optimizadores y, por tanto, que la sinergia entre SHADE y $\text{eSHADE}_{\text{ls}}$ es efectiva para establecer un esquema evolutivo combinado como motor de búsqueda.

El análisis de la eficiencia de búsqueda de GL-SHADE con respecto a algoritmos del estado del arte se hizo en dos partes. Primero se comparó el rendimiento promedio entre nuestro algoritmo y SHADE-ILS considerando múltiples puntos de control. Cabe mencionar que SHADE-ILS es uno de los mejores algoritmos no basados en descomposición e híbridos que se han propuesto hasta la fecha. Los resultados indicaron que al comienzo del proceso de optimización, SHADE-ILS exhibe un mejor rendimiento promedio que GL-SHADE en la mayoría de los problemas de prueba. No obstante, al finalizar la ejecución, se observa justamente lo contrario. Finalmente, se comparó el desempeño entre GL-SHADE y varios algoritmos que han exhibido un desempeño sobresaliente en las recientes competencias de optimización global a gran escala llevadas a cabo durante el *IEEE Congress on Evolutionary Computation*. La comparativa se llevó a cabo usando los criterios empleados en la competencia del 2019. Los resultados obtenidos indicaron que GL-SHADE es un algoritmo competitivo (obtuvo la mejor posición) y el mejor para resolver problemas traslapables.

En este trabajo existen varios aspectos sobre la nueva propuesta que pueden ser mejorados. Por ejemplo, en el diseño de GL-SHADE se consideró una cantidad predefinida y estática de evaluaciones de la función objetivo para realizar búsqueda global y local por iteración. Una posible mejora a este respecto es añadir un mecanismo que ajuste dinámicamente, es decir, adaptativamente, la cantidad de recursos asignados para exploración y explotación basado en el desempeño previo. Es decir, activar por más tiempo el motor (global o local) que mejor se esté ajustando al problema y a la etapa específica del proceso evolutivo. Respecto a mejoras de construcción, es posible considerar o desarrollar otras variantes de SHADE que mejoren el desempeño general. Por ejemplo, recientemente se ha propuesto una variante mejorada de SHADE llamada CSHADE (ver [18]). Se puede explorar la posibilidad de emplear a CSHADE como motor de búsqueda global y también

es posible explorar otros esquemas de evolución efectivos para realizar búsqueda local que mejoren el rendimiento de $\text{eSHADE}_{\text{ls}}$.

A | Implementación en GPU

En este capítulo se discuten varios detalles sobre la implementación en `CUDA + OpenMP` de la nueva propuesta. En la sección A.1 se introducen algunos conceptos importantes relacionados con el uso de *Graphical Processing Units* (GPUs). En la sección A.2 se describe, mediante diagramas de flujo, la implementación del algoritmo GL-SHADE resaltando claramente las partes que son llevadas a cabo en las GPUs así como los casos particulares en donde se decide activar varios núcleos de la CPU en vez de emplear las GPUs para realizar una actividad. Finalmente, en la sección A.3 se exhibe una parte del código correspondiente a la implementación con el fin de mostrar cómo está estructurado el programa principal, indicar la configuración de la rejilla adoptada al invocar las funciones núcleo, entre algunos otros detalles que se consideran importantes.

A.1. Introducción

¿Qué es y para qué se usa una GPU?

Una GPU o unidad de procesamiento gráfico es un dispositivo dotado con una arquitectura que le permite ejecutar cierto tipo de tareas que generalmente serían menos eficientes de realizar si fueran llevadas a cabo por la unidad de procesamiento central (CPU) permitiendo aligerar la carga de trabajo de esta última (razón por lo que a veces se refieren a una GPU como un co-procesador). Una CPU se caracteriza por incorporar una arquitectura adecuada para actividades de propósito general, en contraste con la arquitectura de una GPU la cual es genuinamente útil para procedimientos numéricos paralelizables y, por tanto, ideal para ejecutar algoritmos que procesan grandes bloques de datos en paralelo. Generalmente una CPU integra unos cuantos núcleos de procesamiento mientras que una GPU incorpora decenas, cientos e incluso miles de ellos los cuales están agrupados en múltiples bloques conocidos como multiprocesadores (ver figura A.1 [1]).

En un comienzo, las GPUs eran considerados dispositivos de procesamiento específico ya que su uso estaba restringido a operaciones gráficas. Su tarea consistía en procesar los datos provenientes de la CPU y luego transformarlos en información representable en un

[1] Imagen extraída de [46].

dispositivo de salida (monitor, proyector, etc). Afortunadamente, hoy en día se cuenta con GPUs que pueden ser programadas para realizar otro tipo de tareas. Es decir, existen GPUs de propósito general (GPGPUs) lo que significa que toda la potencia computacional que integran puede ser aprovechada para resolver un problema en particular, generalmente de índole científica: minería de datos, aprendizaje de máquina, compresión de datos, optimización, entre otras [47].

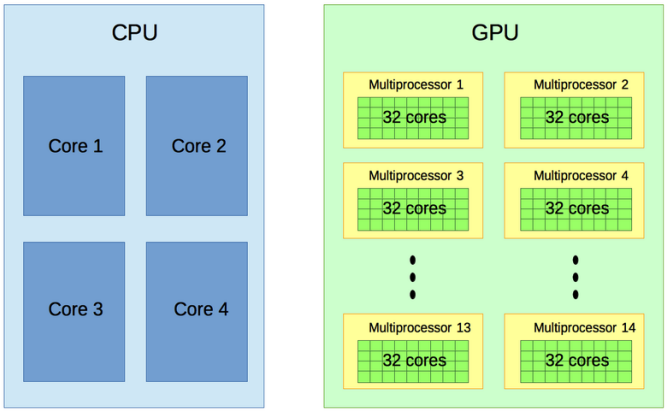

Figura A.1: Comparación del número de núcleos integrados en una CPU y una GPU.

¿Cómo desarrollar programas de propósito general para una GPU?

Para simplificar el desarrollo de los programas GPGPU, varios proveedores han introducido lenguajes de programación, bibliotecas y herramientas para crear código paralelo rápidamente. La plataforma paralela de cómputo basada en GPU y la interfaz de programación de aplicaciones (API, por sus siglas en inglés) desarrollada por Nvidia llamada CUDA (*Computer Unified Device Architecture*) es una de las tecnologías más ampliamente empleadas para el desarrollo de rutinas en GPUs [2] [47]. El lenguaje de programación `C/CUDA` es una extensión del lenguaje `C` que permite el desarrollo del siguiente tipo de rutinas empleando una sintaxis similar a `C`:

Definición A.1.1. Función tipo núcleo: es una rutina ejecutada exclusivamente por la GPU (dispositivo) e invocada exclusivamente por la CPU (anfitrión). El tipo de dato que regresa esta función siempre es vacío.

[2] Desafortunadamente, solo es posible emplear GPUs tecnológicamente habilitadas con CUDA, es decir, aquellas fabricadas por Nvidia.

Definición A.1.2. Función tipo anfitrión: es una rutina invocada y ejecutada exclusivamente por la CPU, es decir, son funciones tradicionales de `C`. Puede regresar cualquier tipo de dato.

Definición A.1.3. Función tipo dispositivo: es una rutina invocada y ejecutada exclusivamente por la GPU. Puede regresar cualquier tipo de dato.

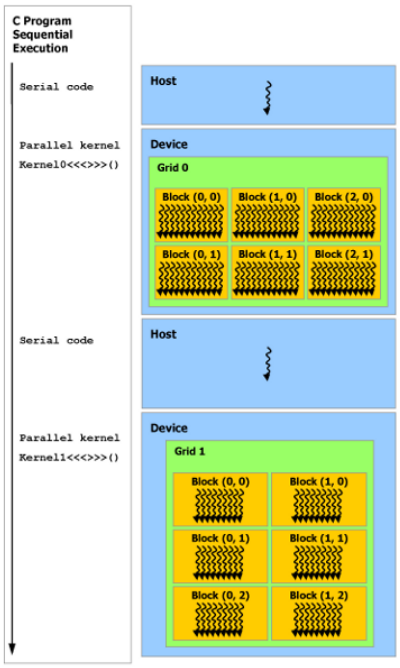

Figura A.2: Ejecución de un programa heterogéneo.

Cada función núcleo (*kernel*) define instrucciones que se ejecutan por muchos hilos (*threads*) al mismo tiempo, siguiendo el modelo **una sola instrucción múltiples datos** (SIMD, por sus siglas en inglés), es decir, se ejecutan múltiples copias de la misma función (una por hilo) [47, 48]. El conjunto de hilos solicitados para ejecutar una función núcleo se llama rejilla (*grid*). A su vez, el conjunto de hilos de una rejilla se divide en varios subconjuntos o grupos de hilos, cada uno de los cuales es denominado bloque (*block*) [3] [49, 47]. En la figura A.2 [4] se esboza la ejecución de un programa heterogéneo (ejecutado tanto en el procesador como en el co-procesador). La ejecución del programa comienza siempre en el anfitrión siguiendo un hilo de ejecución secuencial. Esto se mantiene hasta que se encuentra una función núcleo (en este caso, *kernel* 0) e inmediatamente comienza

[3] Se mencionan los nombres *block*, *kernel*, etc. acorde con la terminología CUDA para que el lector se familiarice con los nombres oficiales y así pueda identificarlos más fácilmente a la hora de programar o leer código.

[4] Imagen extraída de [48].

un hilo de ejecución paralelo en el dispositivo donde, tal como puede observarse, se genera una cuadrilla compuesta por varios bloques los cuales a su vez están compuestos de varios hilos (en nuestro ejemplo se tienen 6 bloques). Cuando el dispositivo ha terminado su tarea, regresa el control al anfitrión y así el hilo de ejecución secuencial continúa hasta encontrar más funciones núcleo (en caso de existir) o el fin del programa.

Es importante notar la diferencia de las configuraciones de las rejillas correspondientes a las funciones núcleo 0 y 1. Ambas tienen 6 bloques establecidos matricialmente pero el número de renglones y columnas es distinto. Al invocar una función núcleo es posible establecer la configuración de la rejilla como se desee. Por ejemplo, en lugar de establecer una estructura matricial de los bloques podemos establecer una lineal e incluso una tridimensional (arreglo de matrices, es decir, un *voxel*).

Similarmente, la configuración de un bloque también es manejable y ahora la estructura de los hilos puede ser establecida: 1D, 2D o 3D. La configuración de la rejilla y los bloques de una función núcleo depende del problema y de cómo quiere resolverse. Para los lectores poco familiarizados con el lenguaje `C/CUDA` y por tanto el uso de *kernels* se sugiere revisar los capítulo 3 y 4 (introductorios) del libro ***CUDA by example*** (ver [50]).

Jerarquías de memoria en una GPU

Todas las GPUs tecnológicamente habilitadas con CUDA incorporan diferentes tipos de memoria con diferentes propiedades: latencia, alcance, tiempo de vida y capacidad, por mencionar algunas. A continuación se enlistan y describen los tipos de memorias más importantes:

- **Memoria de registro:** las variables escalares que se declaran en el ámbito de una función núcleo y no están precedidas por una palabra reservada especial se almacenan en la memoria de registro de forma predeterminada. El acceso a la memoria de registro es **muy rápido**, pero el número de registros disponibles por bloque es limitado.
  Los arreglos que se declaran en la función núcleo también se almacenan en la memoria de registro, pero solo si el acceso a los elementos de la matriz se realiza utilizando índices constantes (significa que el índice que se está utilizando para acceder a un elemento en la matriz no es una variable).
  Las variables de registro son **privadas** por lo que cada hilo tiene su propia copia y ésta existirá mientras exista el hilo. Una vez que el hilo finaliza la ejecución, no se puede volver a acceder a una variable de registro. Las variables declaradas en la memoria de registro pueden leerse y escribirse dentro del núcleo y no se necesita sincronización ya que cada hilo tiene su zona privada de memoria.

- **Memoria compartida:** las variables que están precedidas por la palabra reservada __shared__ se almacenan en la memoria compartida (caché). El acceso a la memoria compartida es rápido (significativamente más rápido que el acceso a la memoria

global pero más lento que el acceso a un registro). No obstante, cada bloque tiene una cantidad limitada de memoria compartida.
La memoria compartida debe declararse dentro del alcance de la función núcleo, pero tiene una vida útil a nivel del bloque (a diferencia del registro o memoria local que tiene una vida útil a nivel de hilo). Cuando se termina la ejecución de un bloque, no se puede acceder a la memoria compartida que se definió en el núcleo.
La memoria compartida puede leerse y escribirse en el núcleo pero su modificación debe sincronizarse a menos que se garantice que cada hilo accederá exclusivamente a una zona de la memoria que otros hilos del bloque no podrán leer ni escribir.

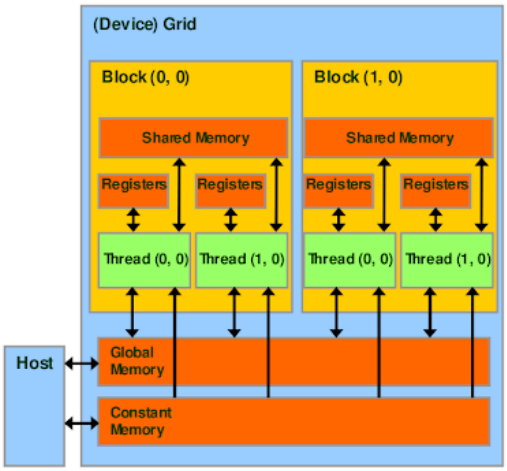

Figura A.3: Modelo de memoria en CUDA.

- **Memoria local:** cualquier variable que no quepa en el espacio de registro permitido para el núcleo se extenderá a la memoria local. La memoria local tiene la misma velocidad de acceso que la memoria global (la cual es lenta comparada con el acceso a registros o memoria compartida).
  Al igual que los registros, la memoria local es privada para el hilo. Cada hilo debe inicializar el contenido de una variable almacenada en la memoria local antes de usarse. No puede confiarse en otro hilo (incluso de su mismo bloque) para inicializar la memoria local porque es privada a nivel de hilo.

- **Memoria global:** las variables que están precedidas por la palabra reservada __device__ se declaran en el alcance global (fuera del alcance de la función núcleo) y son almacenadas en la memoria global. **El acceso a la memoria global es lento. No obstante, se dispone de mucha más memoria global que de memoria compartida.**
  La memoria global tiene una vida útil a nivel de aplicación y es accesible por todos los hilos de todos los núcleos (existen GPUs que permiten la ejecución concurrente

de varias funciones núcleo). Hay que tener cuidado al leer y escribir en la memoria global porque la ejecución de los hilos no se puede sincronizar entre diferentes bloques. La única forma de garantizar que el acceso a la memoria global esté sincronizado es realizando invocaciones de *kernel* separadas (dividiendo el problema en diferentes *kernels* y sincronizando sus invocaciones por medio del anfitrión, es decir, la CPU).

- **Memoria constante:** las variables que están precedidas por la palabra reservada __constant__ se declaran en memoria constante. Al igual que las variables globales, las variables constantes tienen un alcance global. Además, las variables constantes comparten los mismos bancos de memoria que la memoria global (memoria del dispositivo), pero a diferencia de esta última, solo se puede declarar (usar) una cantidad limitada de memoria constante.
  El acceso a la memoria constante es considerablemente más rápido que el acceso a la memoria global porque la memoria constante se almacena en caché y, a diferencia de la memoria global, no se puede escribir en la memoria constante desde una función núcleo. En conclusión, los accesos a memoria constante son rápidos y de gran alcance, pero esta memoria es exclusivamente de lectura.

En la figura A.3 se muestra el modelo de memoria adoptado en CUDA y en la figura A.4 se indican los tres tipos de alcance que se tienen: nivel de hilo (memoria de registro y local), nivel de bloque (memoria compartida) y nivel de dispositivo (memoria global y constante). Saber distinguir los beneficios y limitantes de cada uno de los tipos de memoria es fundamental para explotar al máximo las capacidades una GPU. Para los lectores no familiarizados con el uso de memoria compartida se sugiere leer el capítulo 5, sección 3 del libro ***CUDA by example*** (ver [50]). Ahí es posible encontrar un ejemplo sencillo (producto punto) útil para aprender a usar y aprovechar correctamente la memoria compartida.

A.2. Paralelización de GL-SHADE

En el algoritmo de GL-SHADE (descrito en la sección 5.2) existen varias secciones que pueden ser ejecutadas eficientemente en la GPU, principalmente porque es una propuesta basada en el algoritmo de evolución diferencial el cual puede ser implementado correctamente empleando CUDA [49, 47]. No obstante, en este caso se tienen un par de dificultades:

1. Los algoritmos de SHADE y $\text{eSHADE}_{\text{ls}}$ requieren mayor participación de generadores de números aleatorios (RNGs, por sus siglas en inglés) a diferencia de la ED clásica. La tarea de trabajar con funciones que generan números aleatorios en un contexto paralelo no es sencillo en algunos casos.

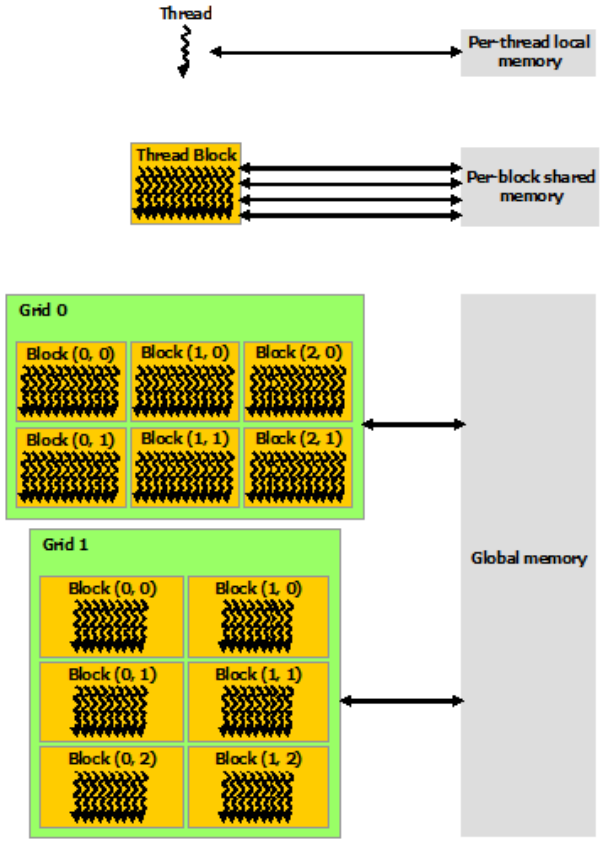

Figura A.4: Alcance de los tipos de memoria.

2. Los procedimientos de búsqueda local que integran la nueva propuesta no pueden ser paralelizados. Esto es un problema sobre todo por el método de perturbación que integra el algoritmo de $\text{eSHADE}_{\text{ls}}$.

Para tratar el primer inconveniente se propone establecer dos fuentes generadoras de números aleatorios, una en la CPU y otra en la GPU. Dependiendo de las circunstancias, se usa una u otra. El segundo inconveniente no puede ser tratado totalmente ya que alterar el flujo de operación puede deteriorar el motor de búsqueda. Por lo tanto, se propone desarrollar una versión multinúcleo de la función objetivo, es decir, activar al menos 2 núcleos de la CPU (se usa OpenMP) para acelerar el proceso de evaluar la aptitud de un nuevo candidato a solución al trabajar con métodos de búsqueda local. Algunos lectores se pueden estar preguntando ¿Por qué no desarrollar una versión CUDA de la función objetivo? En realidad, esto es poco efectivo tratándose de un método de búsqueda local debido a que el costo acumulado de transferencia de datos entre la CPU y GPU no justifica la posible ganancia en velocidad de procesar (evaluar) únicamente a un individuo por iteración. Es así como la versión CUDA de la función objetivo es implementada única-

mente para los métodos de búsqueda basados en población donde en una carga y descarga de datos (transferencia) es posible procesar toda una población justificándose así el costo acumulado de transferencia entre iteraciones o generaciones.

Acorde con lo anterior, en la figura A.5 se muestra un diagrama que resume el flujo de operación correspondiente a la implementación en CUDA de la nueva propuesta. La primera tarea es establecer los parámetros requeridos por GL-SHADE y justo enseguida se ejecutan dos funciones concurrentemente para aleatorizar o iniciar el RNG en el anfitrión (color azul) y en el dispositivo (color rojo). Cuando la función anfitrión **aleatorizar** termina, a la CPU no le importa si la GPU ha terminado con la función núcleo **aleatorizar** ya que inmediatamente procede a definir los componentes instanciando las clases SHADE, $\text{eSHADE}_{\text{ls}}$ y MTS-LS1 para generar los objetos DE1, DE2 y LS1, respectivamente.

Una vez definidos los componentes se da comienzo a la fase de inicialización. Se puede observar la creación de dos poblaciones empleando el método `inicializar_población()` el cual es una función tipo anfitrión definida para ambos objetos DE1 y DE2. Nótese que dicho método realiza varias actividades (algunas concurrentemente) entre las cuales se tienen dos llamadas al dispositivo: **crear población** (se emplea el RNG de la GPU) y **evaluar población** (empleando la versión CUDA de la función objetivo).

Posteriormente, el individuo `mejor_global` (el mejor individuo generado considerando ambas poblaciones) es afinado empleando el método `mejorar()` del objeto LS1. Obsérvese cómo dicho método no realiza ninguna llamada al dispositivo ya que se emplea la versión mutinúcleo de la función objetivo para evaluar a las posibles soluciones candidatas que surgen durante el proceso de mejora (búsqueda local).

El siguiente y último paso corresponde a la fase evolutiva donde los objetos DE1 y DE2 invocan a sus respectivos métodos evolutivos. No obstante, se hace notar que la evolución de las poblaciones es llevada a cabo en secuencia, es decir, primero se evoluciona una y hasta que esta última termina, comienza la otra. En este caso, ambos métodos **evolucionar** realizan tres llamadas al dispositivo: mutación y recombinación, evaluación de descendientes y calcular la media ponderada aritmética (WA) y de Lehmer (WL) sobre los historiales de éxito. Es importante observar cómo el individuo `mejor_global` es constantemente actualizado (internamente).

Antes de continuar y presentar parte del código correspondiente a la implementación de GL-SHADE, se exhibe el flujo de operación de los métodos **evolucionar** implementados para las clases SHADE y $\text{eSHADE}_{\text{ls}}$. En este caso sólo se presenta el diagrama de flujo del proceso evolutivo implementado en CUDA correspondiente al algoritmo de $\text{eSHADE}_{\text{ls}}$ (ver figura A.6), pero el diagrama de SHADE es prácticamente igual con la diferencia de que el procedimiento en color azul-verdoso es suprimido.

Respecto a la figura A.6, la primera tarea es integrar al individuo entrante `mejor_global` a la población actual: en el caso de $\text{eSHADE}_{\text{ls}}$ se sustituye a un miembro seleccionado aleatoriamente y en el caso de SHADE se sustituye siempre al mejor.

Respecto a la fase iterativa, al comienzo de una generación se lleva a cabo un proceso preliminar donde se pre-generan varios datos (F, Cr, p_{best}, etc.) y preparan varias estruc-

turas necesarias para llevar a cabo los procedimiento de mutación y recombinación en el dispositivo: aquí se hace uso del RNG de la CPU. Posteriormente, se ejecutan en secuencia las funciones núcleo **evolucionar población** y **evaluar población de hijos**: la tarea del primer núcleo es generar una población de descendientes mediante los operadores evolutivos de mutación y recombinación, y la del segundo núcleo es calcular la aptitud de cada uno de ellos. Una vez que el último núcleo termina, se descarga la población de descendientes para comenzar con el procedimiento de selección (en la CPU), entre otras varias actividades (ver figura A.6).

Una vez generada la nueva población, dos procedimientos se realizan concurrentemente: mientras el dispositivo calcula la media ponderada aritmética y de Lehmer sobre los historiales `S_Cr` y `S_F`, respectivamente: el anfitrión actualiza el mejor individuo de la nueva población y le da mantenimiento al archivo externo (garantizar que $|A| \leq NP$). Cuando la GPU termina de calcular los promedios, entonces la CPU actualiza las memorias de valores evolutivos `M_Cr` y `M_F` acordemente. En el caso de SHADE, no queda nada más por hacer y se pasa a la siguiente generación (suponiendo que no se cumple la condición de paro). No obstante, en el caso de $\text{eSHADE}_{\text{ls}}$ se procede a invocar la función anfitrión `EDE_LS()` la cual, además del método `mejorar()`, no hace llamadas al dispositivo en ningún momento ya que emplea la versión multinúcleo de la función objetivo para evaluar la viabilidad de los posibles candidatos que van surgiendo durante el proceso de búsqueda.

Vale la pena mencionar que las implementaciones de SHADE y $\text{eSHADE}_{\text{ls}}$ emplean las mismas funciones núcleo **evaluar población de hijos** y **calcular promedios**, pero la función núcleo **evolucionar población** es distinta (porque incorporan estrategias y tipos de recombinación diferentes) y por eso mismo la función anfitrión **preparar datos y estructuras evolutivas** también es distinta. Es importante tener esto en cuenta.

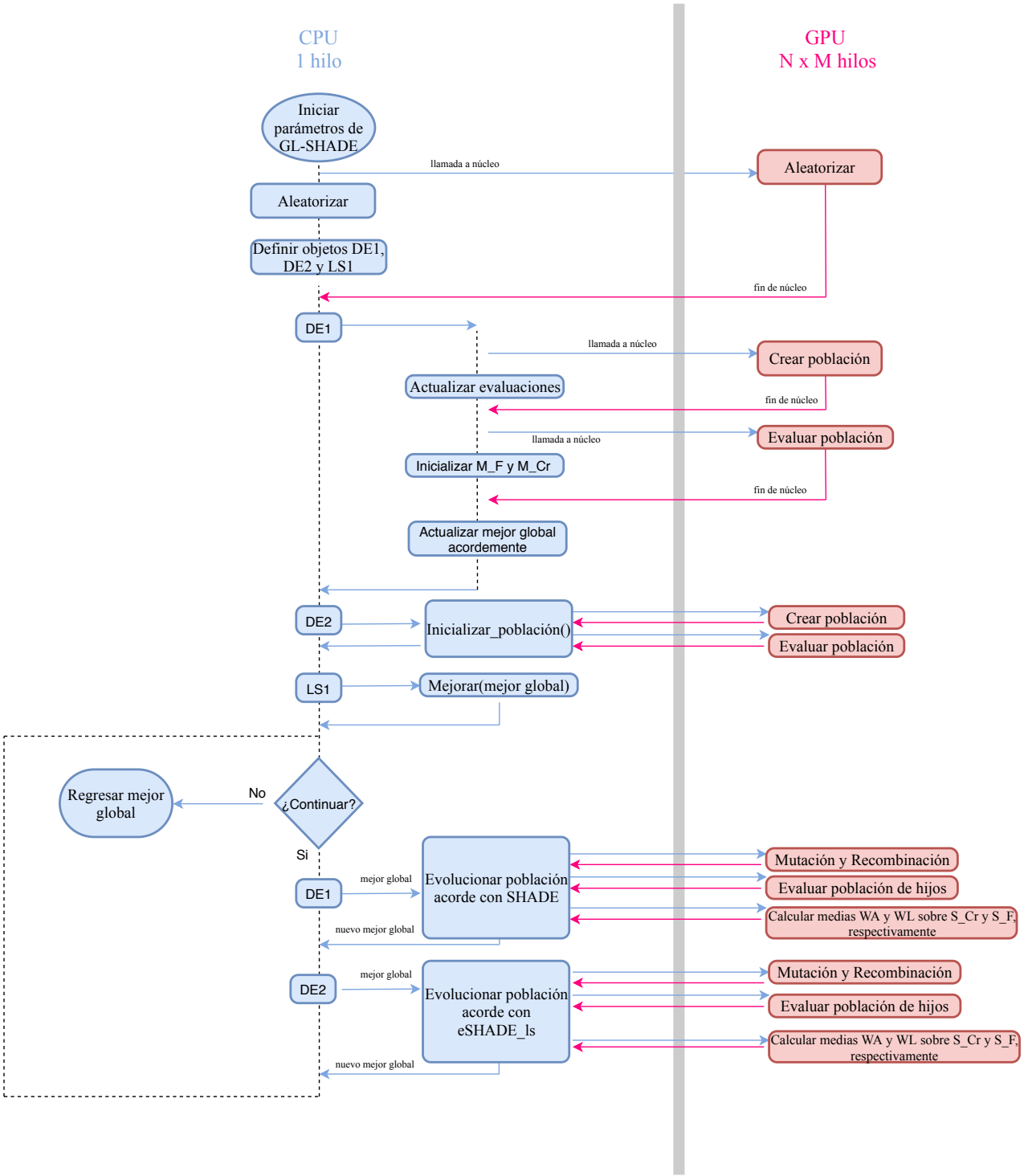

Figura A.5: Diagrama de flujo de la implementación de GL-SHADE empleando CUDA.

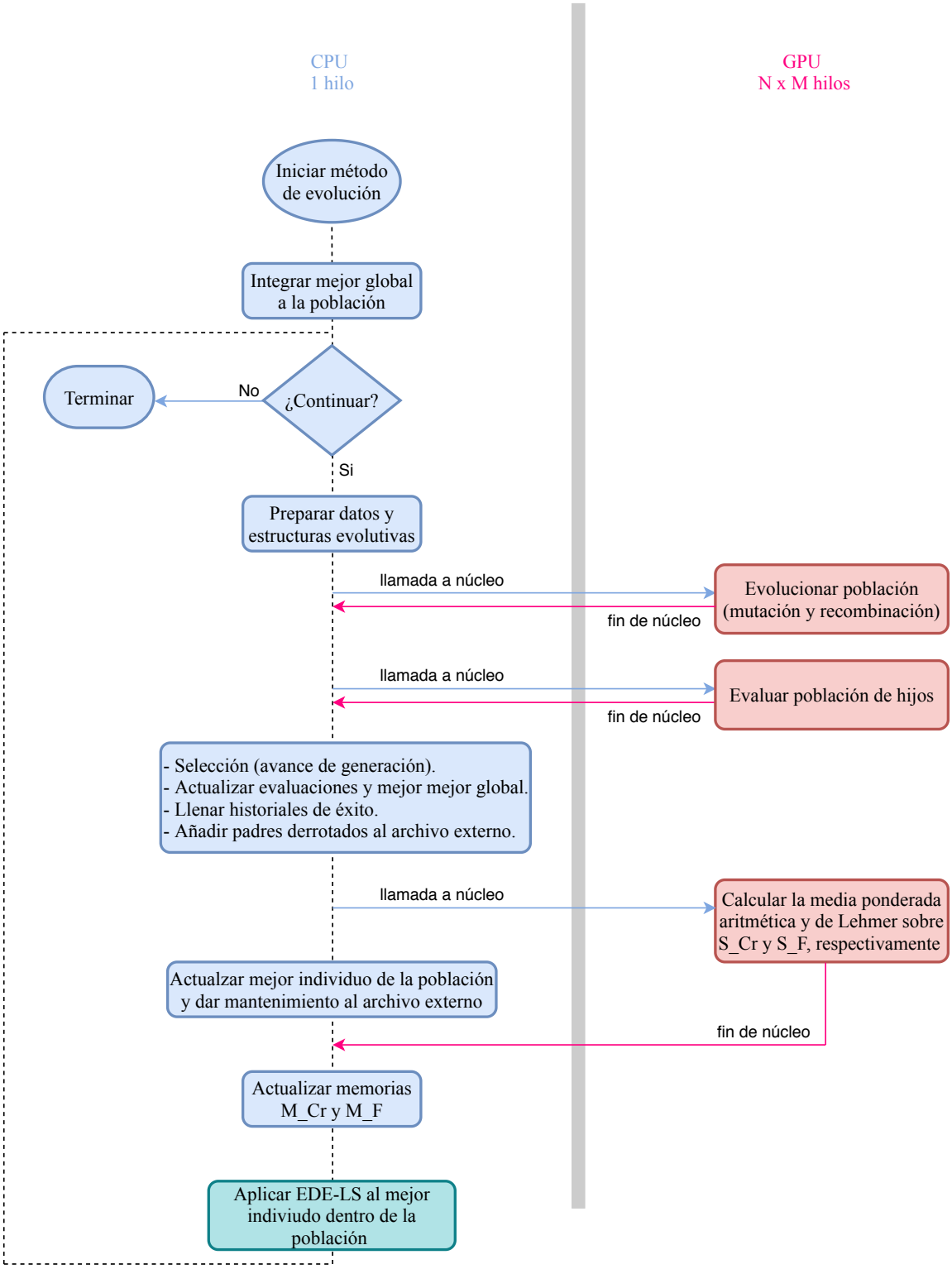

Figura A.6: Diagrama de flujo de la implementación de $\text{eSHADE}_{\text{ls}}$ empleando CUDA.

A.3. Código

En el listado A.1 se muestra el programa principal correspondiente a la implementación del algoritmo GL-SHADE. Algunas partes han sido recortadas con el fin de mostrar menos código, pero sin perder la esencia de la implementación. A primera vista, el código puede parecer voluminoso pero en realidad el núcleo (lo más relevante) de la implementación se encuentra en las líneas 54-93.

Previo al comienzo del algoritmo se define un conjunto de encabezados además de múltiples constantes y tipos de datos personalizados (líneas 1-23). Primero se incluyen varios encabezados normales, después los datos globales y finalmente los encabezados personalizados (líneas 17-23). Estos últimos son la base de toda la implementación (CEC'13 LSGO + GL-SHADE). Dejando a un lado el archivo `objective_function.h`, el resto de encabezados son el soporte de la implementación GL-SHADE. El encabezado `kernel.cu` incluye todas las funciones núcleo requeridas por los métodos (implementados en las clases) que invocan una rutina que ha de ejecutarse en el dispositivo. Por tanto, los archivos `shade.cu`, `eshade_ls.cu` y `mts_ls1.cpp` dependen funcionalmente del encabezado `kernel.cu`.

Finalmente, el encabezado `objective_function.h` es vital para que pueda definirse la función objetivo en sus dos implementaciones (`CUDA` y `OpenMP`) denominadas `F_D` y `F_H`, respectivamente. Ambas son funciones, pero `F_D` es de tipo núcleo y `F_H` de tipo anfitrión. Dichas funciones están pensadas para tener un alcance global permitiendo a cualquier método acceder a ellas. En consecuencia, el encabezado `objective_function.h` incluye todas las funciones, datos y tipos de datos personalizados para poder establecer las funciones `F_D` y `F_H` acorde con cualquiera de los 15 problemas incluidos en el conjunto de prueba CEC'13 LSGO; la función objetivo es seleccionada en tiempo de compilación.

Al comienzo del programa principal (listado A.1, línea 28) se verifica que todos los argumentos de entrada (el tamaño de las poblaciones y valor de la semilla para el generador de números aleatorios) estén dentro de los límites permitidos (ver línea 33). Enseguida se invoca un procedimiento especial que configura el problema de prueba establecido en tiempo de compilación como la función objetivo (línea 38). La fase preliminar termina inicializando el generador de número aleatorios tanto en la CPU como en la GPU así como estableciendo los archivos de salida donde son impresos los reportes de desempeño (líneas 41-53).

Nótese cómo el procedimiento `randomize_gpu()` (línea 44) es una función núcleo. Se establece una rejilla de `N_blocks` bloques, cada uno de los cuales está compuesto de `N_threads` hilos. Como argumentos de entrada se proporcionan una semilla (un número entero arbitrario) y un arreglo de estados (notar que existe una entrada por cada hilo de la rejilla establecida al invocar `randomize_gpu()`). En el listado A.2 se exhibe la función núcleo para inicializar el motor de números aleatorios en la GPU: a cada hilo de la cuadrilla se le asigna un identificador único (definido como una variable privada) la cual es empleada para inicializar la entrada del arreglo de estados (reside en memoria global) acorde con su identificador.

En este caso, todas las funciones núcleo son ejecutadas con una configuración de rejilla lineal, es decir, una rejilla es un conjunto de bloques dispuestos linealmente (un renglón donde hay tantas columnas como bloques) donde cada bloque es un conjunto de hilos también establecidos linealmente. Se decidió adoptar dicha configuración ya que se considera beneficioso al momento de procesar un conjunto de individuos (ver figura A.7[5]) tal como se sugiere en [47].

```
1  ////////////////////////////////NORMAL HEADERS
       ///////////////////////////////////
2      A bunch of headers are included here
3  ////////////////////////CONSTANTS AND OUTPUT FILES
       ///////////////////////////////
4  //readable by both host and device
5  #define PI (3.14159265358979323846264338327 9)
6  #define E  (2.71828182845904523536028747135 2)
7  const int N_threads = 256; //number of threads [CUDA]
8  const int N_blocks = 32; //number of blocks [CUDA]
9  #define maxThreads 6 //max number of cpu cores activated [OMP]
10 const int dim = 1000; //objective function dimension
11 FILE *file_results; //output file: for recording best solution so far by
       checkpoint.
12 FILE *file_report;  //output file: fot recording the evolutionary
       history.
13 struct evol_data_struct{double Cr,F; int j_rand,a,b,p_best;};//used by:
       shade
14 struct evol_data_struct2{double F,Cr; int a,b,p_best;int Jrand,Jend;};//
       used by: eshadels
15 struct rank_ind {int id;double fitness;};//used by: shade/eshadels
16 ////////////////////////////////MY HEADERS
       ///////////////////////////////////
17 #include "rand.c" //Useful functions for generate random number
18 #include "cec13/objective_function.h" //Includes all the data and
       functions needed for running the kernel of the objective function
19 #include "kernel.cu" //Kernel functions
20 #include "shade.cu" //SHADE class implemented here
21 #include "eshade_ls.cu" //eSHADE_ls class implemented here
22 #include "mts_ls1.cpp" //MTS-LS1 class implemented here
23 #include "error_handler.c"
24
25 int main(int argc, char const *argv[])
26 {
27     high_resolution_clock::time_point t1 = high_resolution_clock::now();
       //start timer
28   ////////////////////////////////PRELIMINARIES
       ///////////////////////////////////
29     //Code to test that 3 input arguments were given
30         {}
```

[5]Figura extraída de [47].

```
31     /*Define population sizes and random number generator (rng) seed*/
32     int NP1,NP2; //problem id and popsizes
33     check_and_set_popsize(argv,1,NP1); check_and_set_popsize(argv,2,NP2)
    ; check_and_set_Rseed(argv,3,Rseed);
34
35     /* Define objetive function kernel F_D (implementation of the
    objective function on device) and set bounds accordingly. F_D can be
    used at any scope.
36     F_H is the OMP version of F_D which have the same scope that the
    latter.*/
37     float lb,ub; //bounds
38     set_objective_function(lb,ub); //<- objective_function.h
39
40     /*RNG */
41     int gpuseed = int(Rseed*100); int cpuseed = int(Rseed*10);
42     curandState *state_D;//it stores the rng state in device
43     cudaMalloc(&state_D,N_blocks*N_threads*sizeof(curandState));
44     randomize_gpu<<<N_blocks,N_threads>>>(gpuseed,state_D); //start up
    gpu rng
45     randomize(); //start up cpu rng
46     default_random_engine rng (cpuseed); //rng state in host
47
48     /*Output files to record results.*/
49     string file_results_name = "glshade_results_"+ to_string(NP1) + "
    pop1_" + to_string(NP2) + "pop2_f" + to_string(ID) + ".csv";
50     string file_report_name = "glshade_report_"+ to_string(NP1) + "pop1_
    " + to_string(NP2) + "pop2_f" + to_string(ID) + ".txt";
51
52     //Code to open and initilize output files
53         {}
54   ////////////////////////////////START ALGORITHM
     ////////////////////////////////////
55   /*Set components and gl-shade control parameters*/
56     double before,after;
57     ind global_best;  // best candidate to solution; ind data type is
     declared in objective_function.h
58     int stop_criterion = 3000000;
59     int G_FEs = 25000; int L_FEs = 25000; int current_FEs = 0; int it =
    0;
60     shade DE1(lb,ub,NP1,100,dim,G_FEs); //the 4th argument is H_max1
61     eshade_ls DE2(lb,ub,NP2,100,dim,L_FEs); //the 4th argument is H_max2
62     mts_ls1 LS1(lb,ub,dim,L_FEs);
63
64   /*Initialization*/
65   DE1.init_population_in_device(state_D,global_best,current_FEs); //
     population 1
66   DE2.init_population_in_device(state_D,global_best,current_FEs); //
     population 2
67     //Perform initial report
```

```
68          {}
69        before = global_best.fx;
70      LS1.enhance(global_best,current_FEs,stop_criterion); //enhance best
        candidate to solution
71        after = global_best.fx;
72        //Perform report after early local search has been done
73          {}
74      /*Searching engine*/
75      while (current_FEs < stop_criterion)
76      {
77        //Update iteration counter
78        it++;
79          //Record fitness at the beginning of global exploration
80          before = global_best.fx;
81          //Apply a specialized global search evolution scheme
82          DE1.evolve_in_device(rng,state_D,global_best,current_FEs,
        stop_criterion);
83          //Apply a specialized local search evolution scheme
84          DE2.evolve_in_device(rng,state_D,global_best,current_FEs,
        stop_criterion);
85          //Record fitness at the end of local search
86          after = global_best.fx;
87          /******************* Report ************************/
88          fprintf(file_report,"**************************** Iteration %d
        ****************************\n",it);
89          fprintf(file_report,"\t\tImprovement %d to %d ====> %.3lf %%\n",
        it-1,it,100*(before-after)/before);
90          fprintf(file_report,"\t\tglobal_best ====> f(X):%.4e\n",
        global_best.fx);
91          fprintf(file_report,"\t\tFEs status ====> %d\n",current_FEs);
92      }
93      ////////////////////////////////THE CLOSING
        ////////////////////////////////
94        high_resolution_clock::time_point t2 = high_resolution_clock::now();
          //stop timer
95        auto duration = duration_cast<seconds>( t2 - t1 ).count();
96        /* Final report */
97          //Perform final report
98        printf("Execution Time: %ld sec\n",duration);
99        //Close output
100       fclose (file_results); fclose(file_report);
101       //free space
102       cudaFree(state_D); DE1.free_memory(); DE2.free_memory();
103       free_objective_function_data(); //<- objective_function.h
104     return 0;
105 }
```

Listado A.1: Programa principal

```
1 /***************** Randomize on GPU kernel *********************/
```

```
2 __global__ void randomize_gpu(int GPUseed,curandState *state)
3 { // this function starts random number generator engine on gpu.
4   int randState_tid = blockIdx.x*blockDim.x + threadIdx.x; //
     randState_tid is a private variable
5   curand_init(GPUseed,randState_tid,0,&state[randState_tid]); //*state
     is stored in global memory
6 }
```

Listado A.2: Función núcleo para inicializar motor de números aleatorios en la GPU

Una vez culminada la fase preliminar (ver listado A.1, línea 53) se procede a establecerse los principales parámetros de control (líneas 56-59) y componentes del algoritmo GL-SHADE donde es posible notar la instanciación de las clases `shade`, `eshade_ls` y `mts_ls1` (líneas 60-62). Los componentes basados en población requieren de varios parámetros tales como el tamaño de la población, las restricciones de límite, el tamaño de las memorias `M_CR` y `M_F`, la dimensión del problema y el número de evaluaciones solicitadas por aplicación.

Una vez establecidos los componentes, se procede a inicializar las poblaciones (líneas 60-65), definir como `global_best` al mejor candidato a solución generado (definido internamente por los métodos `init_population_in_device()`) y, finalmente, afinar a la mejor solución global empleando el método `enhance()` (ver línea 70).

```
1   void init_population_in_device(curandState *state_D,ind &global_best,
     int &glshade_current_FEs)
2   {
3     int i;
4
5     // 1. allocate memory
6     cudaMalloc(&pop_D,NP*sizeof(ind));
7
8     //3. Initializing population on device while updating current
     evaluations on host
9     init_population<<<N_blocks,N_threads>>>(state_D,pop_D,NP);
10    glshade_current_FEs += NP;
11
12    //4. Evaluating the created population on device while initializing
     M_F and M_Cr on host
13    F_D<<<N_blocks,N_threads>>>(Ovector_D,mem_D,Pvector_D,r25_D,r50_D,
     r100_D,s_D,w_D,OvectorVec_D,pop_D,NP);
14    for (i = 0; i < H_maxsize; ++i) {M_F[i] = 0.5; M_Cr[i] = 0.5;} //
     Init Cr and F storage
15
16    //5. copy data from device to host
17    cudaMemcpy(pop,pop_D,NP*sizeof(ind),cudaMemcpyDefault);
18
19    //6. free memory
20    cudaFree(pop_D);
21
22    //7. update best and record it
```

```
23      update_best();
24      global_best = pop[best]; //set the best individual of population 1
        as global best
25    }
```

Listado A.3: Procedimiento para crear un población empleando el dispositivo

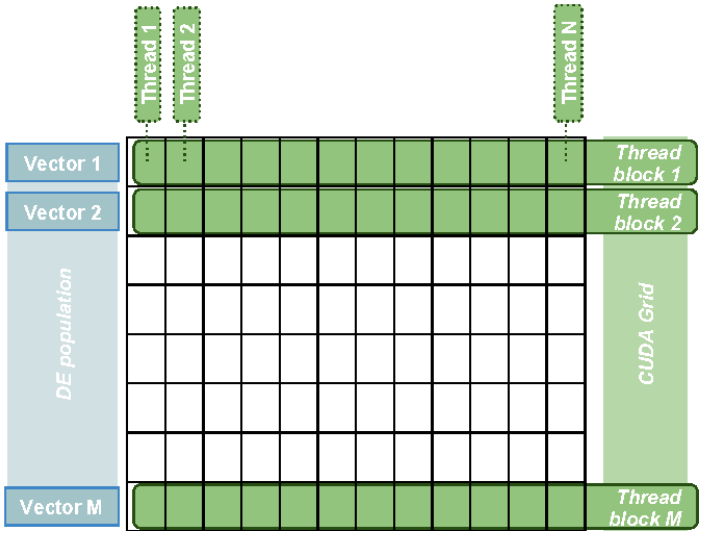

Figura A.7: Configuración de rejilla lineal para procesar una población.

En el listado A.3 se muestra el procedimiento que implementa `DE1` (ver línea 65) para inicializar una población empleando la GPU. La mayor parte del trabajo es llevada a cabo por las funciones núcleo de las líneas 9 y 13. Mientras la GPU está creando una población (línea 9), la CPU actualiza el número de evaluaciones realizadas hasta el momento (línea 10) y luego espera hasta que termine de ejecutarse la función núcleo `init_population()`. Una vez que la GPU ha terminado, la CPU lanza la siguiente función núcleo (línea 13) y, sin esperar a que esta última termine, inicializa las memorias correspondientes (línea 14). Cuando la GPU termina su ejecución, se descargan los datos (línea 17) y entonces la CPU termina el procedimiento definiendo a `global_best` como el mejor individuo generado (líneas 23 y 24).

El método `init_population()` implementado para `DE2` (ver listado A.1, línea 66) es prácticamente igual, con la sutil diferencia de que `global_best` es actualizado si y solo sí el mejor individuo generado (población 2) es mejor que el propio `global_best`, es decir, el mejor de la población 1. A continuación se muestra la implementación de la función núcleo `init_population()`:

```
/****************** Init population kernel ********************/
__global__ void init_population(curandState *state,ind *pop,int NP)
{
```

```
4  /*This function, randomly with uniform distribution, initializes a
    given population.
5  An individual is processed by a block and threads within that block
    process that individual's
6  variables.
7  This means that we're adopting a grid which configuration of blocks is
     linear as well as
8  the configuration of the threads within a block*/
9  int i = blockIdx.x; //block identifier (bid)
10 int j = threadIdx.x; // thread identifier (tid) within block
11 int randState_tid = blockIdx.x*blockDim.x + threadIdx.x; //rng state
    tid
12 curandState localState;
13
14 //For every individual do...
15 while (i<NP) //as much individuals as number of blocks are processed
    by iteration
16 {
17   /************************** Init dimension
    ***************************/
18   //For every dimension do...
19   j = threadIdx.x;
20   while (j < dim) //as much variables as number of threads are
    processed by iteration
21   {
22       // Initialize individual i, variable j
23     localState = state[randState_tid];// Copy state to local memory
    for efficiency
24     pop[i].x[j] = lb_d + curand_uniform(&localState)*(ub_d - lb_d);//
    curand_uniform gives a random number between (0.0,1.0]
25     state [randState_tid] = localState;// Copy state back to global
    memory
26     /* blockDim.x is the number of threads per block. When the threads
     of a block have a tabular (matrix) shape: blockDim.x is
27     the number of rows and blockDim.y the number of columns but since
    here we're using a linear configuration, blockDim.x is
28     the only that matters since blockDim.y is zero in this particular
    case*/
29       j += blockDim.x;
30   }
31   __syncthreads();
32   /* gridDim.x is the number of blocks in the grid. When the blocks of
    a grid have a tabular (matrix) shape the total number
33   of blocks is given for gridDim.x*gridDim.y similarly the number of
    threads within a block that have a matrix shape threads distribution
34   is given by blockDim.x*blockDim.y */
35   i += gridDim.x;
36 }
```

```
37 }
```

Listado A.4: Función núcleo para inicializar una población

Respecto a la función `enhance()` (ver listado A.1, línea 70), ésta ejecuta el algoritmo de MTS-LS1 realizando una implementación secuencial como normalmente se haría pero empleando la función `F_H` (implementación multinúcleo con `OpenMP`) para evaluar el valor de un candidato a solución.

Una vez culminada la fase de inicialización (ver listado A.1, línea 73) se está en posición de comenzar con la etapa evolutiva la cual prácticamente es llevada a cabo en sólo 2 renglones de código (ver listado A.1, líneas 82 y 84). Por temas de espacio sólo se mostrará parte de la implementación del método `evolve_in_device()` asociado al objeto `DE2` (ver listado A.5). No obstante, la implementación asociada al objeto `DE1` es muy similar.

Del listado A.5, nótese (ver líneas 3-19) cómo el individuo `global_best` es integrado a la población en una posición totalmente aleatoria pero siempre cuidado de no sustituir al mejor dentro de la población actual (línea 5). Posteriormente, se procede a reiniciar el contador de evaluaciones (`counter`) y el tamaño de los historiales (`S_size`) además de asignar espacio en la memoria global del dispositivo para múltiples estructuras de datos (líneas 9-19) requeridas para ejecutar la función núcleo de la línea 69. Al comienzo de una generación (línea 22), se inicializa una memoria que incluye todos los individuos de la población actual además de todos los del archivo externo. Es decir, el conjunto $P \cup A$. Posteriormente, la CPU prepara los datos evolutivos (líneas 39-58) requeridos por los procedimientos de mutación y recombinación. Luego se ordena una estructura poblacional especial (`ranklist`) de menor a mayor aptitud útil para seleccionar rápidamente a cualquiera de los `p_best` individuos que sean requeridos durante el procedimiento de mutación. Finalmente, se cargan todos los datos necesarios al dispositivo (líneas 63-66) para invocar la función núcleo `eshade_ls_engine()` **(presente en el listado A.6)** la cual evoluciona una población acorde con los procedimientos de mutación y recombinación del algoritmo $\text{eSHADE}_{\text{ls}}$. El resultado es almacenado en la estructura poblacional `child_D` para posteriormente evaluar el valor de cada hijo procreado empleando la función núcleo `F_D` (ver línea 70). Cuando la GPU finaliza, se descarga la población de hijos (línea 71) lista para rivalizar por la supervivencia (líneas 74-101). Obsérvese cómo se añaden al historial los valores F_i y Cr_i así como al archivo externo el i-ésimo vector padre cuando el i-ésimo vector hijo es más apto que su progenitor (líneas 81-88). Nótese además cómo se actualiza el mejor candidato a solución `global_best`, en caso de ser necesario (línea 90).

Previo a terminar la generación se lleva a cabo la actualización de las memorias `M_F` y `M_Cr`. Mientras la GPU calcula la media ponderada aritmética y la media ponderada de Lehmer sobre los historiales de `S_Cr` y `S_F`, respectivamente (ver líneas 103-109), la CPU actualiza el mejor individuo de la nueva población (línea 111) y le da mantenimiento al archivo externo (ver línea 114). Cuando la GPU termina, la CPU descarga los promedios calculados y actualiza las memorias correspondientes (líneas 116-122).

Finalmente y justo antes de pasar a la siguiente generación se lleva a cabo un proce-

dimiento de perturbación sobre el mejor individuo de la nueva población (ver línea 126). Este procedimiento, al no ser paralelizable, es ejecutado por la CPU. Se emplea la función multinúcleo `F_H` para calcular el valor de un nuevo candidato a solución.

Por último, en el listado A.7 se muestra la función núcleo `shade_engine()` (encargada de evolucionar una población acorde con los procedimientos de mutación y recombinación del algoritmo de SHADE) con el fin de hacer notar las diferencias con respecto a `eshade_ls_engine()`.

Al lector interesado en revisar todos los detalles relacionados con la implementación (incluidos los de las funciones objetivo) se le sugiere revisar el código completo el cual puede ser consultado y/o descargado de: `https://github.com/delmoral313/gl-shade`.

```
1  void evolve_in_device(default_random_engine &rng,curandState *state_D,
    ind &global_best,int &glshade_current_FEs,int glshade_stop_criterion)
2  {
3    //Integrate global_best to population (receive)
4    int r;
5    do r = rnd(0,NP-1); while(r==best); //choose a random position to
    place it
6    pop[r] = global_best; //place it at position r
7    //Set counter and storage size counter
8    counter = 0; S_size = 0; int L,i,j;
9      //Allocate memory on device
10     cudaMalloc(&evol_data_D,NP*sizeof(evol_data_struct2));
11     cudaMalloc(&pop_D,NP*sizeof(ind));
12     cudaMalloc(&child_mu_D,1*sizeof(ind));
13     cudaMalloc(&rank_D,NP*sizeof(rank_ind));
14     cudaMalloc(&memory_D,2*NP*sizeof(ind));
15     cudaMalloc(&child_D,NP*sizeof(ind));
16     cudaMalloc(&S_F_D,NP*sizeof(double));
17     cudaMalloc(&S_Cr_D,NP*sizeof(double));
18     cudaMalloc(&W_D,NP*sizeof(double));
19     cudaMalloc(&mean_D,2*sizeof(double));
20
21     //While stopping condition is not met:
22   while(counter<stop_criterion && glshade_current_FEs<
    glshade_stop_criterion)
23    {
24     /****************** SHADE ************************/
25     // Join Population and external archive
26     memory.insert(memory.end(), &pop[0], &pop[NP]); //memory = pop;
27     memory.insert(memory.end(), A.begin(), A.end());
28
29       //Prepare random data
30       for (i = 0; i < NP; ++i)
31       {
32           ranklist[i].id = i; ranklist[i].fitness = pop[i].fx;
33       /******************Setting F and Cr***********************/
34       // Generate F and Cr using a normal distribution with mean
```

```
35        // taken randomly.
36        r = rnd(0,H_maxsize-1);
37        uF = M_F[r]; normal_distribution<double> Ndistribution_F(uF,0.1)
    ;
38        uCr = M_Cr[r]; normal_distribution<double> Ndistribution_Cr(uCr
    ,0.1);
39
40             evol_data[i].Cr = Ndistribution_Cr(rng);
41             if (evol_data[i].Cr > 1.0) evol_data[i].Cr = 1.0;
42             else if(evol_data[i].Cr < 0.0) evol_data[i].Cr = 0.0;
43
44             evol_data[i].F = Ndistribution_F(rng);
45             if (evol_data[i].F > 1.0) evol_data[i].F = 1.0;
46             while (evol_data[i].F <= 0.0) evol_data[i].F =
    Ndistribution_F(rng);
47             /*******************Setting p_best************************/
48             evol_data[i].p_best = rnd(0,int(rndreal(p_min,p_max)*NP));
    // take an index within best range
49             /*******************Choosing a and b**********************
    */
50             // randomly pick 2 different members
51             do evol_data[i].a = rnd(0,NP-1); while(evol_data[i].a==i);
    // from pop
52             do evol_data[i].b = rnd(0,memory.size()-1); while(evol_data[
    i].b==i || evol_data[i].b==evol_data[i].a); // from pop U archive
53
54             /*******************Get exp crossover window
    ************************/
55             evol_data[i].Jrand = j = rnd(0,D-1);
56             L = 0;
57             do {evol_data[i].Jend = j; j = (j+1)%D; L++;} while(flip(
    evol_data[i].Cr) and L<D);
58        }
59        //Rank population by fitness
60        sort(ranklist);//sort by fitness min => ranklist[0].fitness
61
62      //Load generated data and current population to device
63      cudaMemcpy(evol_data_D,evol_data,NP*sizeof(evol_data_struct2),
    cudaMemcpyDefault);
64      cudaMemcpy(pop_D,pop,NP*sizeof(ind),cudaMemcpyDefault);
65      cudaMemcpy(rank_D,ranklist,NP*sizeof(rank_ind),cudaMemcpyDefault);
66      cudaMemcpy(memory_D,memory.data(),memory.size()*sizeof(ind),
    cudaMemcpyDefault);
67
68      //Lauch kernel: mutation,recombination and function evaluation
69      eshade_ls_engine<<<N_blocks,N_threads>>>(evol_data_D,pop_D,rank_D,
    memory_D,child_D,NP);
70      F_D<<<N_blocks,N_threads>>>(Ovector_D,mem_D,Pvector_D,r25_D,r50_D,
    r100_D,s_D,w_D,OvectorVec_D,child_D,NP);
```

```
71      cudaMemcpy(child,child_D,NP*sizeof(ind),cudaMemcpyDefault);
72
73      //Selection
74      for (i = 0; i < NP; ++i)
75      {
76        //Update FEs counter
77        glshade_current_FEs += 1; counter += 1;
78            if (child[i].fx <= pop[i].fx) // if better than target
    vector then:
79            {
80              //if strictly better then:
81              if (child[i].fx < pop[i].fx)
82              {
83                A_tmp.push_back(pop[i]);//add defeated parent to
    external archive
84                S_F[S_size] = evol_data[i].F;//record F
85                S_Cr[S_size] = evol_data[i].Cr;//record Cr
86                W[S_size] = pop[i].fx - child[i].fx;//record improvement
87                S_size++;//increase storage size counter
88              }
89
90              //update global_best if needed
91              if (child[i].fx<global_best.fx && glshade_current_FEs<=
    glshade_stop_criterion)
92              {
93                global_best = child[i];
94                global_best.FEs_when_found = glshade_current_FEs;
95              }
96              //Advance child to next generation
97              pop[i] = child[i];
98            }
99        if(glshade_current_FEs == some checkpoint)
100         fprintf(file_results,"%d,%d,%.2f,%.6e\n",glshade_current_FEs,
    ID,Rseed,global_best.fx);
101     }
102     //If F and Cr storages are non-empty
103     if (S_size > 0)
104     { //Load F, Cr and W data to device
105       cudaMemcpy(S_F_D,S_F,S_size*sizeof(double),cudaMemcpyDefault);
106       cudaMemcpy(S_Cr_D,S_Cr,S_size*sizeof(double),cudaMemcpyDefault);
107       cudaMemcpy(W_D,W,S_size*sizeof(double),cudaMemcpyDefault);
108       mean_WAWL<<<2,64>>>(S_Cr_D,S_F_D,W_D,S_size,mean_D); //Compute
    mean WA and mean WL in device
109     }
110     //Concurrently update best solution index
111     update_best();
112     // Concurrently check external archive
113     A.insert(A.end(), A_tmp.begin(), A_tmp.end()); // add defeated
    parents to A
```

```
      apply_A_maintenance();//|A| must be less than or equal to popsize
      // Update M_CR and M_F
      if (S_size > 0)
      { //Record means
        cudaMemcpy(mean,mean_D,2*sizeof(double),cudaMemcpyDefault);
        M_F[k] = mean[1]; //weighted Lehmer mean (WL)
        M_Cr[k] = mean[0]; //weighted arithmetic mean (WA)
        k = (k + 1) % H_maxsize;
      }
      // Reset and go again
      S_size = 0; A_tmp.clear(); memory.clear();
      //Apply EDE_LS
      ls_search(D,global_best,glshade_current_FEs,glshade_stop_criterion
   );
    }
      //Free memory
      cudaFree(evol_data_D); cudaFree(pop_D); cudaFree(child_mu_D);
      cudaFree(rank_D); cudaFree(memory_D); cudaFree(child_D); cudaFree(
   S_F_D);
      cudaFree(S_Cr_D); cudaFree(W_D); cudaFree(mean_D);

  }
```

Listado A.5: Evolución de la población 2 acorde con el algoritmo de eSHADE$_{ls}$

```
__global__ void eshade_ls_engine(evol_data_struct2 *dat,ind *pop,
   rank_ind *rank,ind *memory,ind *child,int NP)
{
  /*This function evolves population by applying mutation and
    recombination operators
  according to eshade_ls algorithm. An individual is processed by a
    block and threads
  within that block process that individual variables*/
  int i = blockIdx.x; //block identifier (bid)
  int j = threadIdx.x; // thread identifier (tid) within block
  __shared__ double F; //stored in shared memory
  __shared__ int p_best,a,b,Jrand,Jend; //stored in shared memory

  //For every individual do...
  while (i<NP)
  {
    //Let thread #0 to set evolution data
    if (threadIdx.x == 0)
    {
      F = dat[i].F;
      p_best = dat[i].p_best;
      a = dat[i].a;
      b = dat[i].b;
      Jrand = dat[i].Jrand;
      Jend = dat[i].Jend;
```

```
    }
    __syncthreads(); //barrier: waiting until thread #0 finishes
    /*****************Mutation and exponential crossover both at once
   ************************/
    //For every dimension do...
    j = threadIdx.x;
    while (j<dim)
    { //If within mutation window given by Jrand and Jend
      if((Jend>=Jrand && j>=Jrand && j<=Jend) || (Jend<Jrand && j<=Jend)
    || (Jend<Jrand && j>=Jrand))
      {
        // mutate
            child[i].x[j] = pop[rank[p_best].id].x[j] + F*(pop[a].x[j] -
    memory[b].x[j]);
            // making sure isn't out of boundary
            if (child[i].x[j] > ub_d)
                child[i].x[j] = (ub_d+pop[i].x[j])/2;
            else if (child[i].x[j] < lb_d)
                child[i].x[j] = (lb_d+pop[i].x[j])/2;
        }
      else
        child[i].x[j] = pop[i].x[j];
        j += blockDim.x;
    }
    __syncthreads(); //waiting until all block threads finish to go over
    the next individual
    i += gridDim.x;
  }
}
```

Listado A.6: Función núcleo de eSHADE$_{ls}$ para los procedimientos de mutación y recombinación

```
__global__ void shade_engine(curandState *state,evol_data_struct *dat,
   ind *pop,rank_ind *rank,ind *memory,ind *child,int NP)
{
  /*This function evolves population by applying mutation and
   recombination operators
  according to shade algorithm. An individual is processed by a block
   and threads
  within that block process that individual variables*/
  int i = blockIdx.x; //block identifier (bid)
  int j = threadIdx.x; // thread identifier (tid) within block
  int randState_tid = blockIdx.x*blockDim.x + threadIdx.x;
  curandState localState;
  __shared__ double F,Cr; ////stored in shared memory
  __shared__ int j_rand,p_best,a,b; ////stored in shared memory

  //For every individual do...
  while (i<NP)
```

```
15  {
16    //Let thread #0 to set evolution data
17    if (threadIdx.x == 0)
18    {
19      F = dat[i].F;
20      Cr = dat[i].Cr;
21      p_best = dat[i].p_best;
22      a = dat[i].a;
23      b = dat[i].b;
24      j_rand = dat[i].j_rand;
25    }
26    __syncthreads(); //barrier: waiting until thread #0 finishes
27    /*****************Mutation and binomial crossover both at once
     ************************/
28    //For every dimension do...
29    j = threadIdx.x;
30    while (j < dim)
31    {
32      localState = state[randState_tid]; // load current state
33      if (curand_uniform(&localState)<=Cr || j == j_rand)
34      { // mutate
35            child[i].x[j] = pop[i].x[j] + F*(pop[rank[p_best].id].x[j] -
     pop[i].x[j])
36            + F*(pop[a].x[j] - memory[b].x[j]);
37            // making sure isn't out of boundary
38            if (child[i].x[j] > ub_d)
39                child[i].x[j] = (ub_d+pop[i].x[j])/2;
40            else if (child[i].x[j] < lb_d)
41                child[i].x[j] = (lb_d+pop[i].x[j])/2;
42      }
43      else child[i].x[j] = pop[i].x[j];
44      state [randState_tid] = localState; // update state
45        j += blockDim.x;
46    }
47    __syncthreads(); //waiting until all block threads finish to go over
     the next individual
48    i += gridDim.x;
49  }
50 }
```

Listado A.7: Función núcleo de SHADE para los procedimientos de mutación y recombinación

B | Problemas de prueba

En este apéndice se describen con detalle todas las funciones del conjunto de prueba *IEEE CEC'13 LSGO*. En la sección B.1 se presentan los problemas de prueba por categoría. En la sección B.2 se definen las funciones base a partir de las cuales los problema de prueba están construidos. En la sección B.3 se discute el diseño y las funciones auxiliares empleadas para construir los problemas pertenecientes a las distintas categorías. Finalmente, en la sección B.4 se define cada uno de los problemas de prueba.

El código fuente correspondiente al conjunto de prueba *IEEE CEC'13 LSGO* está disponible en: `http://www.tflsgo.org/special_sessions/cec2019`.

B.1. Categorías

El conjunto de prueba incorpora 15 problemas, los cuales pueden ser clasificados en una de las siguientes cuatro categorías:

1. **Funciones totalmente separables**.

   Definición B.1.1. Una función $f : \mathbb{R}^D \mapsto \mathbb{R}$ es separable si y solo sí [41]:

   $$\arg\min_{\vec{x}} f(\vec{x}) = (\arg\min_{x_1} f(x_1, \cdots), \cdots, \arg\min_{x_D} f(\cdots, x_D)) \tag{B.1}$$

   En otras palabras, una función de D variables es separable si puede reescribirse como una suma de D funciones de una sola variable. Si una función $f(\vec{x})$ es separable, se dice que sus parámetros x_j son independientes.

2. **Funciones parcialmente separables**. Son funciones en las que un pequeño número de variables son dependientes mientras que todas las restantes son independientes [41].

   Definición B.1.2. Una función $f : \mathbb{R}^D \mapsto \mathbb{R}$ es parcialmente separable con m subcomponentes independientes si y solo sí [37]:

   $$\arg\min_{\vec{x}} f(\vec{x}) = (\arg\min_{\vec{x}_1} f(\vec{x}_1, \cdots), \cdots, \arg\min_{\vec{x}_m} f(\cdots, \vec{x}_m)) \tag{B.2}$$

donde $\vec{x}$ es un vector D-dimensional y $\vec{x}_1, \cdots, \vec{x}_m$ son sub-vectores disjuntos de $\vec{x}$ donde $2 \leq m \leq D$. Notar que cuando $m = D$, se tiene una función que es totalmente separable (ver definición B.1.1).

Dentro de esta categoría es posible distinguir dos subclases [37]:

- Funciones parcialmente separables con un conjunto de subcomponentes no separables y un subcomponente completamente separable. Es decir, existe 1 subcomponente separable y $m - 1$ subcomponentes no separables.
- Funciones parcialmente separables con solo un conjunto de subcomponentes no separables y ningún subcomponente completamente separable. Es decir, no existen subcomponentes separables.

3. **Funciones con subcomponentes traslapables**. Los subcomponentes de estas funciones tienen cierto grado de superposición con sus subcomponentes vecinos. Hay dos tipos de funciones superpuestas [37]:

   - **Funciones superpuestas con subcomponentes compatibles:** las variables de decisión compartidas entre dos subcomponentes tienen el mismo valor óptimo con respecto a ambos subcomponentes de la función. En otras palabras, la optimización de un subcomponente puede mejorar el valor del otro subcomponente debido a la optimización de las variables de decisión compartidas.
   - **Funciones superpuestas con subcomponentes en conflicto:** las variables de decisión compartidas tienen un valor óptimo diferente con respecto a cada uno de los subcomponentes de la función. Esto significa que la optimización de un subcomponente puede tener un efecto perjudicial sobre el otro subcomponente superpuesto debido a la naturaleza conflictiva de las variables de decisión compartidas.

4. **Funciones totalmente no separables.**

   Definición B.1.3. Una función $f(\vec{x})$ es completamente no separable si cada par de sus variables de decisión interactúan entre sí [37].

Para la formación de los subcomponentes separables y no separables se emplean 6 funciones base: *Sphere*, *Elliptic*, *Rastrigin*, *Ackley*, *Schwefel* y *Rosenbrock* (todas ellas definidas en la sección B.2). Tomando en cuenta las categorías descritas anteriormente y las funciones base mencionadas, las siguientes 15 funciones de prueba a gran escala son incorporadas en el conjunto de referencia *IEEE CEC'13 LSGO Benchmark*:

1. Funciones totalmente separables

   - f_1: Función *Elliptic*

- f_2: Función *Rastrigin*
- f_3: Función *Ackley*

2. Funciones parcialmente aditivamente separables (ver definición B.1.4)
   - Funciones con un subcomponente separable
     - f_4: Función *Elliptic*
     - f_5: Función *Rastrigin*
     - f_6: Función *Ackley*
     - f_7: Problema de *Schwefel* 1.2
   - Funciones con ningún subcomponente separable
     - f_8: Función *Elliptic*
     - f_9: Función *Rastrigin*
     - f_{10}: Función *Ackley*
     - f_{11}: Problema de *Schwefel* 1.2
3. Funciones traslapables
   - f_{12}: Función de *Rosenbrock*
   - f_{13}: Función de *Schwefel* con subcomponentes superpuestos compatibles
   - f_{14}: Función de *Schwefel* con subcomponentes superpuestos en conflicto
4. Funciones no separables
   - f_{15}: Problema de *Schwefel* 1.2

Definición B.1.4. Una función es parcialmente aditivamente separable si tiene la siguiente forma general [37]:

$$f(\vec{x}) = \sum_{k=1}^{m} f_k(\vec{x}_k) \tag{B.3}$$

donde los $\vec{x}_k$ son sub-vectores de decisión mutuamente excluyentes de $\vec{x}$. Este último es el vector de decisión global D-dimensional y m es el número de subcomponentes independientes. En otras palabras, una función es parcialmente aditivamente separable si puede reescribirse como la suma de m funciones cuyos argumentos de entrada corresponden justamente a cada uno de los k sub-vectores ($\vec{x}_k$) de $\vec{x}$.

B.2. Funciones base

A continuación se definen cada una de las funciones base.

Sphere

$$f_{sphere}(\vec{x}) = \sum_{j=1}^{D} x_j^2 \ ,$$

donde $\vec{x}$ es un vector de decisión D-dimensional. La función *Sphere* es una función bastante sencilla, unimodal y totalmente separable. Por tal motivo, es empleada como un subcomponente completamente separable para algunas de las funciones parcialmente separables.

Elliptic

$$f_{elliptic}(\vec{x}) = \sum_{j=1}^{D} 10^{(6\frac{j-1}{D-1})} x_j^2$$

Rastrigin

$$f_{rastrigin}(\vec{x}) = \sum_{j=1}^{D} [x_j^2 - 10\cos(2\pi x_j) + 10]$$

Ackley

$$f_{ackley}(\vec{x}) = -20\exp\left(-0.2\sqrt{\frac{1}{D}\sum_{j=1}^{D} x_j^2}\right) - \exp\left(\frac{1}{D}\sum_{j=1}^{D}\cos(2\pi x_j)\right) + 20 + e$$

Schwefel

$$f_{schwefel}(\vec{x}) = \sum_{j=1}^{D}\left(\sum_{k=1}^{j} x_j\right)^2$$

Rosenbrock

$$f_{rosenbrock}(\vec{x}) = \sum_{j=1}^{D-1} [100(x_j^2 - x_{j+1})^2 + (x_j - 1)^2]$$

B.3. Diseño

Lo presentado hasta ahora nos permite entender, de forma general, cómo está compuesto el conjunto de referencia. No obstante, para entender los detalles de definición de las funciones de prueba, es necesario introducir conceptos importantes de diseño. El conjunto de referencia *IEEE CEC'13 LSGO Benchmark* es una extensión del conjunto *IEEE CEC'10 LSGO Benchmark* y por tal motivo se añaden nuevas características [37]:

- **El tamaño de los subcomponentes no es uniforme.** Los subcomponentes de un problema de optimización del mundo real tienen muchas probabilidades de ser de tamaños desiguales. Para representar mejor esta característica, las funciones contienen subcomponentes de diferentes tamaños.
- **Desequilibrio en la contribución de subcomponentes.** En muchos problemas del mundo real es probable que los subcomponentes de una función objetivo sean de naturaleza diferente y, por lo tanto, su contribución al valor global puede variar.
- **Funciones con componentes traslapables.** Los subcomponentes de estas funciones tienen cierto grado de superposición con sus subcomponentes vecinos.
- **Nuevas transformaciones a las funciones base:**
  - **Mal condicionamiento.** Se refiere al cuadrado de la relación entre la dirección más grande y la dirección más pequeña de las líneas de contorno. En el caso de una elipsoide, si se estira más en la dirección de un eje que en otras, entonces decimos que la función está mal condicionada.
  - **Ruptura de simetría.** Algunos operadores que generan variaciones genéticas, especialmente aquellos basados en una distribución Gaussiana, son simétricos y, si las funciones también son simétricas, existe un sesgo a favor de estos operadores. Para eliminar tal sesgo, es deseable una transformación de ruptura de simetría.
  - **Irregularidades.** Es deseable introducir cierto grado de irregularidad aplicando alguna transformación.

En la siguiente subsección se introducen algunos símbolos y funciones auxiliares necesarias para entender cómo se construyen las funciones pertenecientes a cada categoría y cómo se incorporan las nuevas características mencionadas.

Símbolos y funciones auxiliares

- $\mathbb{S}$: Es un conjunto que contiene el tamaño de los subcomponentes de una función. Por ejemplo, $\mathbb{S} = \{50, 25, 50, 100\}$ se refiere a una función que tiene $|\mathbb{S}| = 4$ subcomponentes cada uno con 50, 25, 50 y 100 variables de decisión, respectivamente.

- $C_i = \sum_{k=1}^{i} \mathbb{S}_k$: La suma de los primeros i elementos de $\mathbb{S}$. Por conveniencia $C_0 = 0$. C_i es usado para construir el vector de decisión de los diferentes subcomponentes con el tamaño correcto.

- $\mathbb{P}$: Una permutación aleatoria de los índices $\{1, 2, 3..., D\}$.

- w_i : Un peso generado aleatoriamente el cual es empleado como el coeficiente del i-ésimo subcomponente no separable con el fin de generar el efecto de desequilibrio. Los pesos son generados como sigue:

$$w_i = 10^{3N(0,1)} ,$$

  donde $N(0, 1)$ representa una distribución Gaussiana con media = 0 y varianza = 1.

- $\vec{x}_{opt}$: El vector de decisión óptimo para el cual el valor de la función objetivo es mínimo. Éste también es empleado como un vector de desplazamiento para cambiar la ubicación del óptimo global.

- T_{osz} : Una función de transformación para crear irregularidades locales suaves.

$$T_{osz} : \mathbb{R}^D \mapsto \mathbb{R}^D , x_j \to sign(x_j) \cdot \exp(\hat{x}_j + 0.049(\sin(c_1 \hat{x}_j) + \sin(c_2 \hat{x}_j))) , \text{para } j = 1, ..., D$$

$$\text{donde } \hat{x}_j = \begin{cases} \log(|x_j|) & \text{si } x_j \neq 0 \\ 0 & \text{de lo contrario} \end{cases} , \quad sign(x_j) = \begin{cases} -1 & \text{si } x_j < 0 \\ 0 & \text{si } x_j = 0 \\ 1 & \text{si } x_j > 0 \end{cases}$$

$$c_1 = \begin{cases} 10 & \text{si } x_j > 0 \\ 5.5 & \text{de lo contrario} \end{cases} , \quad c_2 = \begin{cases} 7.9 & \text{si } x_j > 0 \\ 3.1 & \text{de lo contrario} \end{cases}$$

- T^{β}_{asy} : Una función de transformación para romper la simetría de las funciones simétricas.

$$T^{\beta}_{asy} : \mathbb{R}^D \mapsto \mathbb{R}^D , x_j \to \begin{cases} x_j^{\left(1+\beta \frac{j-1}{D-1} \sqrt{x_j}\right)} & \text{si } x_j > 0 \\ x_j & \text{de lo contrario} \end{cases} , \text{para } j = 1, ..., D$$

- Λ^{α} : Es una matriz diagonal D-dimensional cuyos elementos en la diagonal se definen acorde con $\lambda_{j,j} = \alpha^{(0.5 \frac{j-1}{D-1})}$. Esta matriz se usa para crear mal condicionamiento. El parámetro α es el número de condición.

- $\mathbf{R}$: Es una matriz de rotación ortogonal la cual es empleada para rotar aleatoriamente el paisaje de la función alrededor de varios ejes.

- m : El tamaño de superposición entre subcomponentes.

- $\vec{1}$: Un vector de 1s.

El diseño de las funciones que pertenecen a las categorías separable y no separable es bastante sencillo; consiste en simplemente someter al vector de decisión $\vec{x}$ a varias operaciones (desplazamiento y algunas transformaciones) para luego emplear dicho vector de decisión resultante $\vec{z}$ como argumento de entrada para alguna de las funciones base que se emplean en este caso. No obstante, el diseño de las funciones pertenecientes a las demás categorías es un poco más elaborado y por tal motivo se discute en las siguientes subsecciones.

Diseño de funciones parcialmente separables

Este tipo de funciones presentan la siguiente forma general [37]:

$$f(\vec{x}) = \left(\sum_{i=1}^{|\mathbb{S}|-1} w_i f_{nonsep}(\vec{z}_i) \right) + f_{sep}(\vec{z}_{|\mathbb{S}|}) \ ,$$

donde w_i es un peso creado aleatoriamente con el fin de crear el efecto de desequilibro y f_{sep} puede ser cualquiera de las siguientes funciones: *Sphere*, la versión no rotada de *Rastrigin* o la versión no rotada de *Ackley*. Para crear una versión no separable (*nonsep*) de estas funciones, puede emplearse una matriz de rotación. El vector $\vec{z}$ es formado al transformar, desplazar y finalmente reordenar las variables del vector $\vec{x}$. Una transformación se ve de la siguiente forma:

$$\vec{y} = \Lambda^{10} T_{asy}^{0.2}(T_{osz}(\vec{x} - \vec{x}_{opt}))$$
$$\vec{z}_i = \vec{y}\left(\mathbb{P}_{[C_{i-1}+1]} \ : \ \mathbb{P}_{[C_i]}\right) \ ,$$

donde el vector $\vec{x}_{opt}$ es empleado para desplazar la ubicación del óptimo global , el conjunto de permutación $\mathbb{P}$ es empleado para reordenar las variables de decisión y C_i es empleado para construir cada uno de los sub-vectores o subcomponentes ($\vec{z}_i$) con el tamaño correspondiente ($\mathbb{S}_i$) especificado en el conjunto $\mathbb{S}$.

Diseño de funciones traslapables con subcomponentes compatibles

El diseño de este tipo de funciones es muy similar al de las funciones parcialmente separables excepto por la formación del vector $\vec{z}_i$ la cual se lleva a cabo como se muestra a continuación:

$$\vec{z}_i = \vec{y}\left(\mathbb{P}_{[C_{i-1}-(i-1)m+1]} \ : \ \mathbb{P}_{[C_i-(i-1)m]}\right)$$

El parámetro m provoca que dos subcomponentes adyacentes tengan en común m variables de decisión. Este parámetro es ajustable y puede variar en el intervalo $1 \leq m \leq \min\{\mathbb{S}\}$. El número total de variables de decisión para este tipo de funciones se calcula de la siguiente forma:

$$D = \left(\sum_{i=1}^{|\mathbb{S}|} \mathbb{S}_i \right) - (m \cdot (|\mathbb{S}| - 1))$$

Diseño de funciones traslapables con subcomponentes en conflicto

El diseño de este tipo de funciones también es muy similar al de las funciones parcialmente separables excepto por la formación del vector $\vec{z}_i$ la cual se lleva a cabo como se muestra a continuación:

$$\vec{y}_i = \vec{x}\ (\mathbb{P}_{[C_{i-1}-(i-1)m+1]}\ :\ \mathbb{P}_{[C_i-(i-1)m]}) - \vec{x}_{opt_i}\ ,$$

$$\vec{z}_i = \Lambda^{10} T^{0.2}_{asy}(T_{osz}(\vec{y}_i))$$

Como se puede observar, cada subcomponente $\vec{z}_i$ es desplazado con un vector diferente $\vec{x}_{opt_i}$. Esto genera un conflicto entre el valor óptimo de las variables de decisión compartidas de dos subcomponentes superpuestos.

B.4. Definición de los problemas de prueba

Funciones totalmente separables

f_1 : Función *Elliptic* desplazada

$$f_1(\vec{z}) = \sum_{j=1}^{D} 10^{(6\frac{j-1}{D-1})} z_j^2$$

- $\vec{z} = T_{osz}(\vec{x} - \vec{x}_{opt})$
- $\vec{x} \in [-100, 100]^D$
- Óptimo global: $f_1(\vec{x}_{opt}) = 0$
- **Propiedades**
  - Unimodal
  - Separable
  - Desplazada
  - Irregularidades locales suaves
  - Mal condicionamiento (número de condición $\approx 10^6$)

f_2 **: Función *Rastrigin* desplazada**

$$f_2(\vec{z}) = \sum_{j=1}^{D} [z_j^2 - 10\cos(2\pi z_j) + 10]$$

- $\vec{z} = \Lambda^{10} T_{asy}^{0.2}(T_{osz}(\vec{x} - \vec{x}_{opt}))$
- $\vec{x} \in [-5, 5]^D$
- Óptimo global: $f_2(\vec{x}_{opt}) = 0$
- **Propiedades**
  - Multimodal
  - Separable
  - Desplazada
  - Irregularidades locales suaves
  - Mal condicionamiento (número de condición ≈ 10)

f_3 **: Función *Ackley* desplazada**

$$f_3(\vec{z}) = -20\exp\left(-0.2\sqrt{\frac{1}{D}\sum_{j=1}^{D} z_j^2}\right) - \exp\left(\frac{1}{D}\sum_{j=1}^{D}\cos(2\pi z_j)\right) + 20 + e$$

- $\vec{z} = \Lambda^{10} T_{asy}^{0.2}(T_{osz}(\vec{x} - \vec{x}_{opt}))$
- $\vec{x} \in [-32, 32]^D$
- Óptimo global: $f_3(\vec{x}_{opt}) = 0$
- **Propiedades**: mismas que f_2

Primer grupo de funciones parcialmente aditivamente separables

f_4 **: Función *Elliptic* desplazada, rotada, 7 no separable y 1 separable**

$$f_4(\vec{z}) = \left(\sum_{i=1}^{|\mathbb{S}|-1} w_i f_{elliptic}(\vec{z}_i)\right) + f_{elliptic}(\vec{z}_{|\mathbb{S}|})$$

- $\mathbb{S} = \{50, 25, 25, 100, 50, 25, 25, 700\}$
- $D = \sum_{i=1}^{|\mathbb{S}|} \mathbb{S}_i = 1000$
- $\vec{y} = \vec{x} - \vec{x}_{opt}$
- $\vec{y}_i = \vec{y}\ (\mathbb{P}_{[C_{i-1}+1]}\ :\ \mathbb{P}_{[C_i]})\ ,\ i \in \{1, ..., |\mathbb{S}|\}$
- $\vec{z}_i = T_{osz}(\mathbf{R}_i \vec{y}_i),\ i \in \{1, ..., |\mathbb{S}| - 1\}$
- $\vec{z}_{|\mathbb{S}|} = T_{osz}(\vec{y}_{|\mathbb{S}|})$
- $\mathbf{R}_i$: una matriz de rotación de tamaño $\mathbb{S}_i \times \mathbb{S}_i$
- $\vec{x} \in [-100, 100]^D$
- Óptimo global: $f_4(\vec{x}_{opt}) = 0$
- **Propiedades**
  - Unimodal
  - Parcialmente separable
  - Desplazada
  - Irregularidades locales suaves
  - Mal condicionamiento (número de condición $\approx 10^6$)

f_5 : Función *Rastrigin* desplazada, rotada, 7 no separable y 1 separable

$$f_5(\vec{z}) = \left(\sum_{i=1}^{|\mathbb{S}|-1} w_i f_{rastrigin}(\vec{z}_i) \right) + f_{rastrigin}(\vec{z}_{|\mathbb{S}|})$$

- $\mathbb{S} = \{50, 25, 25, 100, 50, 25, 25, 700\}$
- $D = \sum_{i=1}^{|\mathbb{S}|} \mathbb{S}_i = 1000$
- $\vec{y} = \vec{x} - \vec{x}_{opt}$
- $\vec{y}_i = \vec{y}\ (\mathbb{P}_{[C_{i-1}+1]}\ :\ \mathbb{P}_{[C_i]})\ ,\ i \in \{1, ..., |\mathbb{S}|\}$
- $\vec{z}_i = \Lambda^{10} T^{0.2}_{asy}(T_{osz}(\mathbf{R}_i \vec{y}_i)),\ i \in \{1, ..., |\mathbb{S}| - 1\}$
- $\vec{z}_{|\mathbb{S}|} = \Lambda^{10} T^{0.2}_{asy}(T_{osz}(\vec{y}_{|\mathbb{S}|}))$
- $\mathbf{R}_i$: una matriz de rotación de tamaño $\mathbb{S}_i \times \mathbb{S}_i$

- $\vec{x} \in [-5, 5]^D$
- Óptimo global: $f_5(\vec{x}_{opt}) = 0$
- **Propiedades**
  - Multimodal
  - Parcialmente separable
  - Desplazada
  - Irregularidades locales suaves
  - Mal condicionamiento (número de condición ≈ 10)

f_6 : Función *Ackley* desplazada, rotada, 7 no separable y 1 separable

$$f_6(\vec{z}) = \left(\sum_{i=1}^{|\mathbb{S}|-1} w_i f_{ackley}(\vec{z}_i) \right) + f_{ackley}(\vec{z}_{|\mathbb{S}|})$$

- $\mathbb{S} = \{50, 25, 25, 100, 50, 25, 25, 700\}$
- $D = \sum_{i=1}^{|\mathbb{S}|} \mathbb{S}_i = 1000$
- $\vec{y} = \vec{x} - \vec{x}_{opt}$
- $\vec{y}_i = \vec{y}\ (\mathbb{P}_{[C_{i-1}+1]}\ :\ \mathbb{P}_{[C_i]})\ ,\ i \in \{1, ..., |\mathbb{S}|\}$
- $\vec{z}_i = \Lambda^{10} T^{0.2}_{asy}(T_{osz}(\mathbf{R}_i \vec{y}_i)),\ i \in \{1, ..., |\mathbb{S}| - 1\}$
- $\vec{z}_{|\mathbb{S}|} = \Lambda^{10} T^{0.2}_{asy}(T_{osz}(\vec{y}_{|\mathbb{S}|}))$
- $\mathbf{R}_i$: una matriz de rotación de tamaño $\mathbb{S}_i \times \mathbb{S}_i$
- $\vec{x} \in [-32, 32]^D$
- Óptimo global: $f_6(\vec{x}_{opt}) = 0$
- **Propiedades** : mismas que f_5

f_7 : Función *Schwefel* desplazada, rotada, 7 no separable y 1 separable

$$f_7(\vec{z}) = \left(\sum_{i=1}^{|\mathbb{S}|-1} w_i f_{schwefel}(\vec{z}_i) \right) + f_{sphere}(\vec{z}_{|\mathbb{S}|})$$

- $\mathbb{S} = \{50, 25, 25, 100, 50, 25, 25, 700\}$
- $D = \sum_{i=1}^{|\mathbb{S}|} \mathbb{S}_i = 1000$
- $\vec{y} = \vec{x} - \vec{x}_{opt}$
- $\vec{y}_i = \vec{y}\ (\mathbb{P}_{[C_{i-1}+1]}\ :\ \mathbb{P}_{[C_i]})\ ,\ i \in \{1, ..., |\mathbb{S}|\}$
- $\vec{z}_i = T^{0.2}_{asy}(T_{osz}(\mathbf{R}_i \vec{y}_i)),\ i \in \{1, ..., |\mathbb{S}| - 1\}$
- $\vec{z}_{|\mathbb{S}|} = T^{0.2}_{asy}(T_{osz}(\vec{y}_{|\mathbb{S}|}))$
- $\mathbf{R}_i$: una matriz de rotación de tamaño $\mathbb{S}_i \times \mathbb{S}_i$
- $\vec{x} \in [-100, 100]^D$
- Óptimo global: $f_7(\vec{x}_{opt}) = 0$
- **Propiedades**
  - Multimodal
  - Parcialmente separable
  - Desplazada
  - Irregularidades locales suaves

Segundo grupo de funciones parcialmente aditivamente separables

f_8 : Función *Elliptic* desplazada, rotada y 20 no separable

$$f_8(\vec{z}) = \sum_{i=1}^{|\mathbb{S}|} w_i f_{elliptic}(\vec{z}_i)$$

- $\mathbb{S} = \{50, 50, 25, 25, 100, 100, 25, 25, 50, 25, 100, 25, 100, 50, 25, 25, 25, 100, 50, 25\}$
- $D = \sum_{i=1}^{|\mathbb{S}|} \mathbb{S}_i = 1000$
- $\vec{y} = \vec{x} - \vec{x}_{opt}$

- $\vec{y}_i = \vec{y}\ (\mathbb{P}_{[C_{i-1}+1]}\ :\ \mathbb{P}_{[C_i]})\ ,\ i \in \{1, ..., |\mathbb{S}|\}$
- $\vec{z}_i = T_{osz}(\mathbf{R}_i \vec{y}_i),\ i \in \{1, ..., |\mathbb{S}|\}$
- $\mathbf{R}_i$: una matriz de rotación de tamaño $\mathbb{S}_i \times \mathbb{S}_i$
- $\vec{x} \in [-100, 100]^D$
- Óptimo global: $f_8(\vec{x}_{opt}) = 0$
- **Propiedades**: mismas que f_4

f_9 : Función *Rastrigin* desplazada, rotada y 20 no separable

$$f_9(\vec{z}) = \sum_{i=1}^{|\mathbb{S}|} w_i f_{rastrigin}(\vec{z}_i)$$

- $\mathbb{S} = \{50, 50, 25, 25, 100, 100, 25, 25, 50, 25, 100, 25, 100, 50, 25, 25, 25, 100, 50, 25\}$
- $D = \sum_{i=1}^{|\mathbb{S}|} \mathbb{S}_i = 1000$
- $\vec{y} = \vec{x} - \vec{x}_{opt}$
- $\vec{y}_i = \vec{y}\ (\mathbb{P}_{[C_{i-1}+1]}\ :\ \mathbb{P}_{[C_i]})\ ,\ i \in \{1, ..., |\mathbb{S}|\}$
- $\vec{z}_i = \Lambda^{10} T^{0.2}_{asy}(T_{osz}(\mathbf{R}_i \vec{y}_i)),\ i \in \{1, ..., |\mathbb{S}|\}$
- $\mathbf{R}_i$: una matriz de rotación de tamaño $\mathbb{S}_i \times \mathbb{S}_i$
- $\vec{x} \in [-5, 5]^D$
- Óptimo global: $f_9(\vec{x}_{opt}) = 0$
- **Propiedades**: mismas que f_5

f_{10} : Función *Ackley* desplazada, rotada y 20 no separable

$$f_{10}(\vec{z}) = \sum_{i=1}^{|\mathbb{S}|} w_i f_{ackley}(\vec{z}_i)$$

- $\mathbb{S} = \{50, 50, 25, 25, 100, 100, 25, 25, 50, 25, 100, 25, 100, 50, 25, 25, 25, 100, 50, 25\}$
- $D = \sum_{i=1}^{|\mathbb{S}|} \mathbb{S}_i = 1000$

- $\vec{y} = \vec{x} - \vec{x}_{opt}$
- $\vec{y}_i = \vec{y}\ (\mathbb{P}_{[C_{i-1}+1]}\ :\ \mathbb{P}_{[C_i]})\ ,\ i \in \{1, ..., |\mathbb{S}|\}$
- $\vec{z}_i = \Lambda^{10} T^{0.2}_{asy}(T_{osz}(\mathbf{R}_i \vec{y}_i)),\ i \in \{1, ..., |\mathbb{S}|\}$
- $\mathbf{R}_i$: una matriz de rotación de tamaño $\mathbb{S}_i \times \mathbb{S}_i$
- $\vec{x} \in [-32, 32]^D$
- Óptimo global: $f_{10}(\vec{x}_{opt}) = 0$
- **Propiedades**: mismas que f_6

f_{11} : Función *Schwefel* desplazada, rotada y 20 no separable

$$f_{11}(\vec{z}) = \sum_{i=1}^{|\mathbb{S}|} w_i f_{schwefel}(\vec{z}_i)$$

- $\mathbb{S} = \{50, 50, 25, 25, 100, 100, 25, 25, 50, 25, 100, 25, 100, 50, 25, 25, 25, 100, 50, 25\}$
- $D = \sum_{i=1}^{|\mathbb{S}|} \mathbb{S}_i = 1000$
- $\vec{y} = \vec{x} - \vec{x}_{opt}$
- $\vec{y}_i = \vec{y}\ (\mathbb{P}_{[C_{i-1}+1]}\ :\ \mathbb{P}_{[C_i]})\ ,\ i \in \{1, ..., |\mathbb{S}|\}$
- $\vec{z}_i = T^{0.2}_{asy}(T_{osz}(\mathbf{R}_i \vec{y}_i)),\ i \in \{1, ..., |\mathbb{S}|\}$
- $\mathbf{R}_i$: una matriz de rotación de tamaño $\mathbb{S}_i \times \mathbb{S}_i$
- $\vec{x} \in [-100, 100]^D$
- Óptimo global: $f_{11}(\vec{x}_{opt}) = 0$
- **Propiedades**
  - Unimodal
  - Parcialmente separable
  - Desplazada
  - Irregularidades locales suaves

Funciones traslapables

f_{12} : Función *Rosenbrock* desplazada

$$f_{12}(\vec{z}) = \sum_{j=1}^{D-1}[100(z_j^2 - z_{j+1})^2 + (z_j - 1)^2]$$

- $D = 1000$
- $\vec{z} = \vec{x}$
- $\vec{x} \in [-100, 100]^D$
- Óptimo global: $f_{12}(\vec{x}_{opt} + \vec{1}) = 0$
- **Propiedades**
  - Multimodal
  - Separable
  - Desplazada
  - Irregularidades locales suaves

f_{13} : Función *Schwefel* desplazada con subcomponentes superpuestos compatibles

$$f_{13}(\vec{z}) = \sum_{i=1}^{|\mathbb{S}|} w_i f_{schwefel}(\vec{z}_i)$$

- $\mathbb{S} = \{50, 50, 25, 25, 100, 100, 25, 25, 50, 25, 100, 25, 100, 50, 25, 25, 25, 100, 50, 25\}$
- $C_i = \sum_{k=1}^{i} \mathbb{S}_i$, $C_0 = 0$
- $D = \left(\sum_{i=1}^{|\mathbb{S}|} \mathbb{S}_i\right) - m(|\mathbb{S}| - 1) = 905$
- $\vec{y} = \vec{x} - \vec{x}_{opt}$
- $\vec{y}_i = \vec{y}\ (\mathbb{P}_{[C_{i-1}-(i-1)m+1]}\ :\ \mathbb{P}_{[C_i-(i-1)m]})$, $i \in \{1, ..., |\mathbb{S}|\}$
- $\vec{z}_i = T_{asy}^{0.2}(T_{osz}(\mathbf{R}_i\vec{y}_i))$, $i \in \{1, ..., |\mathbb{S}|\}$
- $m = 5$: tamaño de traslape
- $\mathbf{R}_i$: una matriz de rotación de tamaño $\mathbb{S}_i \times \mathbb{S}_i$

- $\vec{x} \in [-100, 100]^D$
- Óptimo global: $f_{13}(\vec{x}_{opt}) = 0$
- **Propiedades**
  - Unimodal
  - No separable
  - Traslapada
  - Desplazada
  - Irregularidades locales suaves

f_{14} : Función *Schwefel* desplazada con subcomponentes superpuestos en conflicto

$$f_{14}(\vec{z}) = \sum_{i=1}^{|\mathbb{S}|} w_i f_{schwefel}(\vec{z}_i)$$

- $\mathbb{S} = \{50, 50, 25, 25, 100, 100, 25, 25, 50, 25, 100, 25, 100, 50, 25, 25, 25, 100, 50, 25\}$
- $D = \left(\sum_{i=1}^{|\mathbb{S}|} \mathbb{S}_i\right) - m(|\mathbb{S}| - 1) = 905$
- $\vec{y}_i = \vec{x}\ (\mathbb{P}_{[C_{i-1}-(i-1)m+1]}\ :\ \mathbb{P}_{[C_i-(i-1)m]}) - \vec{x}_{opt_i}$
- $\vec{x}_{opt_i}$: es un vector de desplazamiento de tamaño $\mathbb{S}_i$ para el i-ésimo subcomponente
- $\vec{z}_i = T^{0.2}_{asy}(T_{osz}(\mathbf{R}_i\vec{y}_i))$
- $m = 5$: tamaño de traslape
- $\mathbf{R}_i$: una matriz de rotación de tamaño $\mathbb{S}_i \times \mathbb{S}_i$
- $\vec{x} \in [-100, 100]^D$
- Óptimo global: $f_{14}(\vec{x}_{opt}) = 0$
- **Propiedades**
  - Unimodal
  - No separable
  - Subcomponentes en conflicto
  - Desplazada
  - Irregularidades locales suaves

Funciones totalmente no separables

f_{15} : Función *Schwefel* desplazada

$$f_{15}(\vec{z}) = \sum_{j=1}^{D} \left(\sum_{k=1}^{j} z_j \right)^2$$

- $D = 1000$
- $\vec{z} = T_{asy}^{0.2}(T_{osz}(\vec{x} - \vec{x}_{opt}))$
- $\vec{x} \in [-100, 100]^D$
- Óptimo global: $f_{15}(\vec{x}_{opt}) = 0$
- **Propiedades**
  - Unimodal
  - No separable
  - Desplazada
  - Irregularidades locales suaves

C | Resultados numéricos

En este apéndice se presenta un resumen de los resultados obtenidos al calcular el desempeño de los siguientes algoritmos:

- GL-SHADE
- MTS-LS1
- SHADE
- $\text{eSHADE}_{\text{ls}}$
- SHADE-ILS

El desempeño de estos optimizadores es estimado realizando 25 ejecuciones independientes por problema del conjunto de prueba *IEEE CEC'13 LSGO* (ver apéndice B). En este caso, los algoritmos fueron implementados y programados para reportar la mejor solución obtenida al alcanzar un cierto porcentaje del número máximo de evaluaciones de la función objetivo (max_{FEs}) establecido como condición de paro. En la tabla C.1 se muestra un conjunto de puntos de control donde cada uno de ellos corresponde a un cierto porcentaje de max_{FEs} que debe ser alcanzado para reportar el desempeño.

De acuerdo con lo anterior, en la sección C.1 se muestra una tabla por algoritmo donde puede ser consultado el desempeño promedio correspondiente a cada punto de control y función de prueba.

Adicionalmente, en la sección C.2 se muestran los resultados estadísticos (media, mediana, desviación estándar, mejor y peor solución) de cada algoritmo pero considerando sólo un subconjunto de los puntos de control (aquellos resaltados en negritas de la tabla C.1).

C.1. Desempeño promedio por punto de control y función de prueba

Tabla C.1: Puntos de control.

Número	**Punto de control**	**Porcentaje del número máximo de evaluaciones de la función objetivo**
1	**1.2E+05**	**4 %**
2	3.0E+05	10 %
3	**6.0E+05**	**20 %**
4	9.0E+05	30 %
5	1.2E+06	40 %
6	1.5E+06	50 %
7	1.8E+06	60 %
8	2.1E+06	70 %
9	2.4E+06	80 %
10	2.7E+06	90 %
11	**3.0E+06**	**100 %**

Tabla C.2: Desempeño promedio de GL-SHADE por punto de control y función de prueba.

	GL-SHADE										
Función	**4 %**	**10 %**	**20 %**	**30 %**	**40 %**	**50 %**	**60 %**	**70 %**	**80 %**	**90 %**	**100 %**
f_1	1.7842E+05	6.6584E+03	3.8196E+01	5.1575E-03	1.7606E-09	4.8518E-14	2.9254E-17	8.2711E-20	4.4816E-21	7.2777E-22	3.7379E-23
f_2	6.8434E+02	1.8677E+02	2.3512E+01	9.8797E+00	9.7905E+00	9.7195E+00	9.6318E+00	9.4501E+00	8.9570E+00	8.0858E+00	7.7610E+00
f_3	2.0003E+01	2.0001E+01	2.0000E+01	2.0000E+01	2.0000E+01	2.0000E+01	2.0000E+01	2.0000E+01	2.0000E+01	2.0000E+01	2.0000E+01
f_4	5.6297E+10	2.0516E+10	3.4434E+09	8.1204E+08	3.6651E+08	2.0314E+08	1.1776E+08	7.8633E+07	5.7283E+07	4.0650E+07	3.0118E+07
f_5	4.3545E+06	3.1930E+06	2.6669E+06	2.4611E+06	2.4179E+06	2.3316E+06	2.3125E+06	2.2708E+06	2.2634E+06	2.2631E+06	2.2264E+06
f_6	1.0546E+06	1.0542E+06	1.0540E+06	1.0528E+06	1.0470E+06	1.0429E+06	1.0408E+06	1.0390E+06	1.0369E+06	1.0355E+06	1.0342E+06
f_7	1.5022E+09	2.6920E+08	2.8045E+07	2.7785E+06	3.0078E+05	7.6318E+04	1.8245E+04	2.8082E+03	2.9202E+02	2.2502E+01	2.3673E+00
f_8	4.0515E+14	5.4014E+13	1.1049E+13	4.0021E+12	1.6672E+12	8.9353E+11	5.1000E+11	3.1985E+11	2.0563E+11	1.4784E+11	1.1064E+11
f_9	2.5506E+09	2.4178E+09	2.3967E+09	2.3800E+09	2.3696E+09	2.3661E+09	2.3622E+09	2.3620E+09	2.3615E+09	2.3615E+09	2.3615E+09
f_{10}	9.3807E+07	9.3349E+07	9.2747E+07	9.2324E+07	9.2176E+07	9.2049E+07	9.1961E+07	9.1895E+07	9.1823E+07	9.1762E+07	9.1697E+07
f_{11}	9.4476E+11	9.2770E+11	9.2732E+11	9.2729E+11	9.2729E+11	9.2729E+11	9.2729E+11	9.2729E+11	9.2729E+11	9.2729E+11	9.2729E+11
f_{12}	2.4216E+04	4.9790E+03	9.6040E+02	3.3009E+02	1.1611E+02	4.0317E+01	1.2929E+01	7.1369E+00	4.1620E+00	2.9626E+00	3.1896E-01
f_{13}	2.6902E+10	8.4873E+09	2.7300E+09	6.8605E+08	1.5461E+08	2.4297E+07	5.4114E+06	1.6356E+06	4.9455E+05	1.3953E+05	3.9831E+04
f_{14}	3.7086E+11	1.0543E+11	1.4210E+10	3.2474E+08	4.5760E+07	1.9816E+07	1.0547E+07	7.0879E+06	5.7410E+06	5.0997E+06	4.7868E+06
f_{15}	1.1253E+08	7.2411E+07	3.2416E+07	1.4205E+07	8.0694E+06	5.3011E+06	3.7063E+06	2.6891E+06	2.0589E+06	1.6115E+06	1.2919E+06

Tabla C.3: Desempeño promedio de MTS-LS1 por punto de control y función de prueba.

MTS-LS1											
Función	**4 %**	**10 %**	**20 %**	**30 %**	**40 %**	**50 %**	**60 %**	**70 %**	**80 %**	**90 %**	**100 %**
f_1	7.1926E-15	3.0182E-25	3.0182E-25	3.0182E-25	3.0182E-25	3.0182E-25	3.0182E-25	3.0182E-25	3.0182E-25	3.0182E-25	3.0182E-25
f_2	5.7012E+03	4.6326E+03	4.3831E+03	4.2450E+03	4.1845E+03	4.1597E+03	4.1179E+03	4.0977E+03	4.0232E+03	3.9984E+03	3.9680E+03
f_3	2.0008E+01	2.0007E+01	2.0007E+01	2.0007E+01	2.0007E+01	2.0007E+01	2.0007E+01	2.0007E+01	2.0007E+01	2.0007E+01	2.0007E+01
f_4	1.4988E+12	1.3018E+12	1.2212E+12	1.1796E+12	1.1564E+12	1.1381E+12	1.1197E+12	1.1101E+12	1.0996E+12	1.0922E+12	1.0820E+12
f_5	5.9199E+07	5.9149E+07	5.9126E+07	5.9114E+07	5.9107E+07	5.9100E+07	5.9097E+07	5.9094E+07	5.9091E+07	5.9089E+07	5.9088E+07
f_6	1.0513E+06	1.0510E+06	1.0509E+06	1.0508E+06	1.0507E+06	1.0507E+06	1.0507E+06	1.0506E+06	1.0506E+06	1.0506E+06	1.0506E+06
f_7	1.0150E+10	9.2878E+09	8.7338E+09	8.5160E+09	8.3677E+09	8.2729E+09	8.1846E+09	8.0904E+09	8.0348E+09	7.9499E+09	7.9165E+09
f_8	8.2773E+16	6.8888E+16	6.3760E+16	6.1302E+16	5.9622E+16	5.8309E+16	5.7539E+16	5.6752E+16	5.5948E+16	5.5279E+16	5.4924E+16
f_9	4.5931E+10	4.4619E+10	4.4213E+10	4.4013E+10	4.3865E+10	4.3781E+10	4.3702E+10	4.3649E+10	4.3608E+10	4.3569E+10	4.3535E+10
f_{10}	9.4249E+07	9.4230E+07	9.4219E+07	9.4213E+07	9.4210E+07	9.4207E+07	9.4204E+07	9.4202E+07	9.4201E+07	9.4199E+07	9.4198E+07
f_{11}	1.6800E+12	1.3726E+12	1.2801E+12	1.2405E+12	1.2188E+12	1.2014E+12	1.1946E+12	1.1854E+12	1.1742E+12	1.1696E+12	1.1674E+12
f_{12}	2.3143E+03	2.2417E+03	2.2080E+03	2.1918E+03	2.1795E+03	2.1697E+03	2.1623E+03	2.1575E+03	2.1502E+03	2.1455E+03	2.1429E+03
f_{13}	5.2771E+10	4.7764E+10	4.5008E+10	4.3802E+10	4.1921E+10	4.1522E+10	4.1135E+10	4.0813E+10	4.0620E+10	4.0459E+10	4.0259E+10
f_{14}	1.3488E+12	1.2738E+12	1.2339E+12	1.2172E+12	1.1973E+12	1.1851E+12	1.1759E+12	1.1606E+12	1.1524E+12	1.1473E+12	1.1431E+12
f_{15}	3.6603E+08	3.4031E+08	3.2379E+08	3.1611E+08	3.0970E+08	3.0648E+08	3.0273E+08	2.9958E+08	2.9755E+08	2.9651E+08	2.9521E+08

Tabla C.4: Desempeño promedio de SHADE por punto de control y función de prueba.

SHADE											
Función	**4 %**	**10 %**	**20 %**	**30 %**	**40 %**	**50 %**	**60 %**	**70 %**	**80 %**	**90 %**	**100 %**
f_1	4.3294E+08	8.0978E+07	1.7272E+07	8.1972E+06	4.8269E+06	3.7023E+06	3.2493E+06	2.5373E+06	2.3026E+06	2.2196E+06	2.1845E+06
f_2	1.6305E+04	1.5682E+04	1.5473E+04	1.5420E+04	1.5396E+04	1.5388E+04	1.5384E+04	1.5382E+04	1.5381E+04	1.5380E+04	1.5379E+04
f_3	2.1288E+01	2.0964E+01	2.0624E+01	2.0424E+01	2.0298E+01	2.0220E+01	2.0164E+01	2.0125E+01	2.0097E+01	2.0077E+01	2.0060E+01
f_4	5.9577E+10	1.7451E+10	5.9757E+09	3.3179E+09	2.3596E+09	1.9049E+09	1.6080E+09	1.4217E+09	1.2955E+09	1.2087E+09	1.1522E+09
f_5	5.0000E+06	3.5083E+06	2.8134E+06	2.6153E+06	2.5608E+06	2.5127E+06	2.4693E+06	2.4370E+06	2.4095E+06	2.3762E+06	2.3572E+06
f_6	1.0631E+06	1.0615E+06	1.0601E+06	1.0594E+06	1.0589E+06	1.0583E+06	1.0578E+06	1.0572E+06	1.0569E+06	1.0565E+06	1.0564E+06
f_7	2.3728E+08	4.2830E+07	1.0935E+07	6.1918E+06	4.6883E+06	4.0276E+06	3.6577E+06	3.4451E+06	3.3174E+06	3.1803E+06	3.1042E+06
f_8	7.1635E+13	1.3544E+13	4.1950E+12	2.2197E+12	1.5887E+12	1.3143E+12	1.1451E+12	1.0660E+12	9.6763E+11	9.1472E+11	8.9442E+11
f_9	1.9875E+09	1.5609E+09	1.5042E+09	1.4717E+09	1.4499E+09	1.4340E+09	1.4224E+09	1.4106E+09	1.3975E+09	1.3872E+09	1.3792E+09
f_{10}	9.4152E+07	9.3702E+07	9.3474E+07	9.3300E+07	9.3219E+07	9.3112E+07	9.2977E+07	9.2875E+07	9.2849E+07	9.2786E+07	9.2711E+07
f_{11}	9.3844E+11	9.3402E+11	9.3355E+11	9.3352E+11	9.3350E+11	9.3349E+11	9.3349E+11	9.3349E+11	9.3349E+11	9.3340E+11	9.3339E+11
f_{12}	1.6504E+10	1.1383E+09	1.2691E+08	3.6629E+07	2.6414E+07	2.0774E+07	1.3015E+07	1.1479E+07	1.0651E+07	8.7240E+06	8.2921E+06
f_{13}	6.5152E+09	1.9536E+09	7.0836E+08	3.5263E+08	2.0830E+08	1.4852E+08	9.9956E+07	7.8619E+07	7.0579E+07	6.1054E+07	5.7585E+07
f_{14}	4.3136E+10	4.4122E+09	7.3684E+08	3.2991E+08	2.3359E+08	1.9593E+08	1.7906E+08	1.5781E+08	1.4831E+08	1.3300E+08	1.2944E+08
f_{15}	2.2297E+07	9.3188E+06	5.5794E+06	3.9025E+06	3.1185E+06	2.6125E+06	2.2065E+06	1.9328E+06	1.7131E+06	1.5536E+06	1.4229E+06

Tabla C.5: Desempeño promedio de $\text{eSHADE}_{\text{ls}}$ por punto de control y función de prueba.

$\text{eSHADE}_{\text{ls}}$											
Función	**4 %**	**10 %**	**20 %**	**30 %**	**40 %**	**50 %**	**60 %**	**70 %**	**80 %**	**90 %**	**100 %**
f_1	6.6170E+06	4.0930E+05	5.0671E+00	4.1167E-07	2.6146E-08	3.9170E-10	1.2590E-11	7.7415E-13	1.9395E-14	1.2105E-15	4.0663E-17
f_2	2.5071E+02	2.6228E+01	3.3988E+00	3.3829E+00	3.3829E+00	3.3829E+00	3.3829E+00	3.3829E+00	3.3829E+00	3.3829E+00	3.3829E+00
f_3	2.0003E+01	2.0000E+01	2.0000E+01	2.0000E+01	2.0000E+01	2.0000E+01	2.0000E+01	2.0000E+01	2.0000E+01	2.0000E+01	2.0000E+01
f_4	9.8731E+11	5.4987E+10	7.9420E+09	2.9344E+09	1.6638E+09	1.0848E+09	7.7142E+08	5.7930E+08	4.2625E+08	3.1681E+08	2.5036E+08
f_5	2.9640E+07	1.7565E+07	1.0498E+07	1.0247E+07	1.0247E+07	1.0247E+07	1.0247E+07	1.0247E+07	1.0247E+07	1.0247E+07	1.0247E+07
f_6	1.0590E+06	1.0577E+06	1.0562E+06	1.0529E+06	1.0498E+06	1.0464E+06	1.0445E+06	1.0428E+06	1.0416E+06	1.0404E+06	1.0395E+06
f_7	7.4300E+09	2.8555E+09	9.0724E+07	5.7595E+06	4.4093E+05	5.4447E+04	1.5858E+04	5.8067E+03	2.1068E+03	7.6584E+02	2.6734E+02
f_8	5.6340E+16	1.8817E+15	4.1664E+14	2.5383E+14	1.7250E+14	1.1833E+14	8.9982E+13	7.6486E+13	6.6419E+13	5.3141E+13	4.6405E+13
f_9	4.2031E+09	2.9342E+09	2.4750E+09	2.3665E+09	2.3646E+09	2.3640E+09	2.3640E+09	2.3640E+09	2.3640E+09	2.3639E+09	2.3639E+09
f_{10}	9.4347E+07	9.3834E+07	9.3047E+07	9.2828E+07	9.2642E+07	9.2471E+07	9.2340E+07	9.2238E+07	9.2150E+07	9.2072E+07	9.1982E+07
f_{11}	1.0594E+12	9.3935E+11	9.3346E+11	9.3271E+11	9.3260E+11	9.3259E+11	9.3259E+11	9.3258E+11	9.3258E+11	9.3258E+11	9.3258E+11
f_{12}	4.4905E+04	8.2337E+03	3.2014E+03	1.4679E+03	1.0990E+03	8.6950E+02	7.3125E+02	6.1795E+02	5.2829E+02	4.2966E+02	3.6988E+02
f_{13}	3.4874E+10	1.9605E+10	2.5832E+09	8.1917E+08	2.7872E+08	9.5540E+07	2.8054E+07	8.4975E+06	2.9680E+06	1.0181E+06	2.9525E+05
f_{14}	9.8006E+11	5.1710E+11	6.3330E+10	1.2912E+10	2.1374E+09	1.1432E+08	2.0353E+07	1.1764E+07	7.6498E+06	5.7684E+06	4.9800E+06
f_{15}	4.3584E+09	3.5591E+08	3.0700E+07	9.2358E+06	4.6995E+06	2.8978E+06	1.9622E+06	1.4058E+06	1.0260E+06	7.6986E+05	5.8184E+05

Tabla C.6: Desempeño promedio de SHADE-ILS por punto de control y función de prueba.

SHADE-ILS											
Función	**4 %**	**10 %**	**20 %**	**30 %**	**40 %**	**50 %**	**60 %**	**70 %**	**80 %**	**90 %**	**100 %**
f_1	5.1296E+04	1.9509E-07	3.5512E-23	1.4594E-23	8.7605E-24	2.4083E-25	2.4076E-25	5.8109E-28	2.6765E-28	2.6765E-28	2.5558E-28
f_2	2.5452E+03	1.9900E+03	1.7864E+03	1.5908E+03	1.4628E+03	1.3436E+03	1.2628E+03	1.1777E+03	1.1115E+03	1.0763E+03	1.0415E+03
f_3	2.0100E+01	2.0096E+01	2.0088E+01	2.0084E+01	2.0080E+01	2.0076E+01	2.0072E+01	2.0072E+01	2.0068E+01	2.0068E+01	2.0068E+01
f_4	3.5864E+10	1.0362E+10	3.7448E+09	1.9016E+09	1.1648E+09	8.1192E+08	6.2252E+08	5.0252E+08	4.1872E+08	3.4248E+08	3.0128E+08
f_5	2.3556E+06	2.2536E+06	2.1032E+06	1.7980E+06	1.6616E+06	1.5480E+06	1.4936E+06	1.4121E+06	1.3622E+06	1.3422E+06	1.3310E+06
f_6	1.0516E+06	1.0468E+06	1.0424E+06	1.0392E+06	1.0392E+06	1.0376E+06	1.0352E+06	1.0348E+06	1.0328E+06	1.0316E+06	1.0316E+06
f_7	3.6856E+08	4.3860E+07	1.5421E+06	1.9624E+05	7.4444E+04	3.2220E+04	1.3509E+04	5.3824E+03	2.0134E+03	6.7295E+02	2.2356E+02
f_8	2.3631E+14	5.3276E+13	1.4339E+13	7.5382E+12	4.1387E+12	2.6132E+12	1.7116E+12	1.1104E+12	8.9156E+11	7.2832E+11	5.9937E+11
f_9	2.8760E+08	2.7304E+08	2.4932E+08	2.1728E+08	2.0028E+08	1.9236E+08	1.8276E+08	1.7636E+08	1.6628E+08	1.6252E+08	1.5780E+08
f_{10}	9.3956E+07	9.3676E+07	9.3192E+07	9.3036E+07	9.2928E+07	9.2808E+07	9.2764E+07	9.2724E+07	9.2592E+07	9.2572E+07	9.2556E+07
f_{11}	5.5884E+09	5.6088E+08	1.3048E+08	4.7328E+07	1.6407E+07	6.2268E+06	2.8656E+06	1.7042E+06	1.1290E+06	7.7112E+05	5.3888E+05
f_{12}	2.6464E+03	2.0412E+03	1.7688E+03	1.0752E+03	8.3989E+02	5.3170E+02	2.5449E+02	1.6344E+02	1.1663E+02	8.1042E+01	6.4886E+01
f_{13}	1.3623E+10	3.1421E+09	5.6046E+08	1.2080E+08	2.4265E+07	1.0059E+07	5.5428E+06	3.3888E+06	2.1982E+06	1.4949E+06	1.0706E+06
f_{14}	1.8416E+11	3.0338E+10	4.9090E+08	5.7972E+07	2.8908E+07	1.8624E+07	1.3809E+07	1.1187E+07	9.4932E+06	8.3972E+06	7.6280E+06
f_{15}	8.8616E+07	4.5516E+07	1.6893E+07	7.7788E+06	4.2196E+06	2.8944E+06	2.1292E+06	1.6468E+06	1.3044E+06	1.0541E+06	8.6832E+05

C.2. Resumen de resultados estadísticos para los principales puntos de control

Tabla C.7: La media, mediana, desviación estándar, mejor y peor solución obtenida tras realizar 25 ejecuciones de GL-SHADE por función de prueba.

GL-SHADE									
1000D		***f1***	***f2***	***f3***	***f4***	***f5***	***f6***	***f7***	***f8***
	Mejor	1.1570E+05	6.2182E+02	2.0002E+01	2.8042E+10	3.4482E+06	1.0475E+06	5.7466E+08	1.1492E+14
	Mediana	1.6220E+05	6.8721E+02	2.0003E+01	5.6700E+10	4.4502E+06	1.0547E+06	1.4164E+09	3.4428E+14
1.2E+05	Peor	4.6395E+05	7.5821E+02	2.0004E+01	1.1534E+11	5.2045E+06	1.0597E+06	2.8667E+09	1.0190E+15
	Media	1.7842E+05	6.8434E+02	2.0003E+01	5.6297E+10	4.3545E+06	1.0546E+06	1.5022E+09	4.0515E+14
	DevSt	7.2299E+04	3.0459E+01	3.5845E-04	2.0058E+10	5.0855E+05	3.6941E+03	5.8094E+08	2.4962E+14
	Mejor	1.6893E-01	1.5767E+01	2.0000E+01	6.8304E+08	1.9810E+06	1.0472E+06	1.0319E+07	7.8981E+11
	Mediana	2.6790E+01	2.1187E+01	2.0000E+01	2.4396E+09	2.6616E+06	1.0543E+06	2.5514E+07	9.3245E+12
6.0E+05	Peor	2.1990E+02	5.3728E+01	2.0000E+01	1.1301E+10	3.2326E+06	1.0596E+06	5.6731E+07	3.0003E+13
	Media	3.8196E+01	2.3512E+01	2.0000E+01	3.4434E+09	2.6669E+06	1.0540E+06	2.8045E+07	1.1049E+13
	DevSt	4.8843E+01	8.5986E+00	3.1765E-05	2.5806E+09	3.2901E+05	3.6766E+03	1.3186E+07	7.1567E+12
	Mejor	3.1986E-26	0.0000E+00	2.0000E+01	1.1811E+07	1.5633E+06	1.0091E+06	4.3496E-02	5.0130E+09
	Mediana	1.0714E-25	3.9798E+00	2.0000E+01	2.4373E+07	2.2398E+06	1.0357E+06	8.7364E-01	2.2224E+10
3.0E+06	Peor	5.1394E-22	4.1788E+01	2.0000E+01	8.1139E+07	3.0970E+06	1.0532E+06	1.0269E+01	4.2171E+11
	Media	3.7379E-23	7.7610E+00	2.0000E+01	3.0118E+07	2.2264E+06	1.0342E+06	2.3673E+00	1.1064E+11
	DevSt	1.1330E-22	1.0094E+01	0.0000E+00	1.7759E+07	3.6616E+05	1.0735E+04	2.7597E+00	1.4976E+11
1000D		***f9***	***f10***	***f11***	***f12***	***f13***	***f14***	***f15***	
	Mejor	1.5833E+09	9.2531E+07	9.2403E+11	1.9366E+04	1.4494E+10	1.2171E+11	9.2875E+07	
	Mediana	2.5805E+09	9.3932E+07	9.3782E+11	2.4348E+04	2.6208E+10	3.3494E+11	1.1225E+08	
1.2E+05	Peor	4.0948E+09	9.4356E+07	1.0042E+12	3.1865E+04	4.7445E+10	7.5552E+11	1.4355E+08	
	Media	2.5506E+09	9.3807E+07	9.4476E+11	2.4216E+04	2.6902E+10	3.7086E+11	1.1253E+08	
	DevSt	6.6079E+08	4.7359E+05	1.8168E+10	3.1165E+03	6.9505E+09	1.7758E+11	1.2393E+07	
	Mejor	1.3770E+09	9.1844E+07	9.1543E+11	5.2959E+02	1.5592E+09	3.6160E+08	9.3834E+06	
	Mediana	2.4076E+09	9.2739E+07	9.2276E+11	8.3406E+02	2.3997E+09	1.1711E+10	2.9304E+07	
6.0E+05	Peor	4.0324E+09	9.3790E+07	9.4949E+11	2.0717E+03	6.1054E+09	6.9691E+10	7.4877E+07	
	Media	2.3967E+09	9.2747E+07	9.2732E+11	9.6040E+02	2.7300E+09	1.4210E+10	3.2416E+07	
	DevSt	6.6920E+08	5.2362E+05	1.0629E+10	4.1194E+02	1.0123E+09	1.5361E+10	1.9492E+07	
	Mejor	1.3743E+09	9.0954E+07	9.1543E+11	1.3131E-23	1.3388E+04	4.3995E+06	1.2041E+05	
	Mediana	2.4072E+09	9.1637E+07	9.2276E+11	2.2215E-23	2.9806E+04	4.7472E+06	1.1409E+06	
3.0E+06	Peor	4.0217E+09	9.2909E+07	9.4943E+11	3.9866E+00	1.2446E+05	5.3146E+06	3.0018E+06	
	Media	2.3615E+09	9.1697E+07	9.2729E+11	3.1896E-01	3.9831E+04	4.7868E+06	1.2919E+06	
	DevSt	6.6534E+08	4.6953E+05	1.0580E+10	1.1038E+00	2.9867E+04	2.1480E+05	1.1058E+06	

Tabla C.8: La media, mediana, desviación estándar, mejor y peor solución obtenida tras realizar 25 ejecuciones de MTS-LS1 por función de prueba.

MTS-LS1									
1000D		*f1*	*f2*	*f3*	*f4*	*f5*	*f6*	*f7*	*f8*
1.2E+05	Mejor	5.2895E-15	4.3393E+03	2.0000E+01	4.3442E+11	4.0907E+07	1.0430E+06	2.6386E+09	2.7249E+16
	Mediana	6.6665E-15	5.1238E+03	2.0006E+01	1.5660E+12	5.8673E+07	1.0519E+06	8.8948E+09	8.2305E+16
	Peor	1.3663E-14	1.1703E+04	2.0017E+01	3.0125E+12	8.2137E+07	1.0583E+06	2.0047E+10	1.3417E+17
	Media	7.1926E-15	5.7012E+03	2.0008E+01	1.4988E+12	5.9199E+07	1.0513E+06	1.0150E+10	8.2773E+16
	DevSt	2.0117E-15	1.8481E+03	5.5388E-03	7.4202E+11	9.7035E+06	3.9160E+03	5.5899E+09	3.0161E+16
6.0E+05	Mejor	3.6365E-26	3.7367E+03	2.0000E+01	2.3483E+11	4.0877E+07	1.0420E+06	2.4768E+09	1.8843E+16
	Mediana	2.9172E-25	4.2164E+03	2.0006E+01	1.3421E+12	5.8531E+07	1.0516E+06	7.7691E+09	5.9443E+16
	Peor	5.7768E-25	5.3508E+03	2.0017E+01	2.5642E+12	8.2134E+07	1.0580E+06	1.7941E+10	1.1167E+17
	Media	3.0182E-25	4.3831E+03	2.0007E+01	1.2212E+12	5.9126E+07	1.0509E+06	8.7338E+09	6.3760E+16
	DevSt	1.4434E-25	4.8765E+02	5.2813E-03	5.9838E+11	9.6941E+06	3.9988E+03	4.8015E+09	2.8426E+16
3.0E+06	Mejor	3.6365E-26	3.5150E+03	2.0000E+01	1.9308E+11	4.0861E+07	1.0412E+06	2.3535E+09	1.2421E+16
	Mediana	2.9172E-25	3.8661E+03	2.0006E+01	1.1999E+12	5.8477E+07	1.0513E+06	6.6278E+09	4.5851E+16
	Peor	5.7768E-25	4.7098E+03	2.0017E+01	2.3393E+12	8.2133E+07	1.0578E+06	1.6853E+10	9.7530E+16
	Media	3.0182E-25	3.9680E+03	2.0007E+01	1.0820E+12	5.9088E+07	1.0506E+06	7.9165E+09	5.4924E+16
	DevSt	1.4434E-25	3.2003E+02	5.1194E-03	5.4042E+11	9.6923E+06	4.0691E+03	4.4246E+09	2.7031E+16
1000D		*f9*	*f10*	*f11*	*f12*	*f13*	*f14*	*f15*	
1.2E+05	Mejor	2.7245E+10	9.2882E+07	1.0392E+12	5.7554E+02	3.0436E+10	4.5118E+11	1.6645E+08	
	Mediana	4.3638E+10	9.4283E+07	1.6554E+12	1.4029E+03	5.1994E+10	1.3627E+12	2.9881E+08	
	Peor	8.5252E+10	9.5918E+07	3.0267E+12	8.5455E+03	1.0929E+11	2.4274E+12	8.7811E+08	
	Media	4.5931E+10	9.4249E+07	1.6800E+12	2.3143E+03	5.2771E+10	1.3488E+12	3.6603E+08	
	DevSt	1.4099E+10	8.4621E+05	4.3914E+11	2.1843E+03	1.7134E+10	4.4142E+11	1.7848E+08	
6.0E+05	Mejor	2.5693E+10	9.2850E+07	9.4561E+11	5.4364E+02	2.8349E+10	4.3106E+11	1.5314E+08	
	Mediana	4.2175E+10	9.4261E+07	1.2845E+12	1.3844E+03	4.1873E+10	1.1778E+12	2.7176E+08	
	Peor	8.4204E+10	9.5891E+07	1.5363E+12	8.2573E+03	8.5791E+10	2.3037E+12	7.5319E+08	
	Media	4.4213E+10	9.4219E+07	1.2801E+12	2.2080E+03	4.5008E+10	1.2339E+12	3.2379E+08	
	DevSt	1.3252E+10	8.3979E+05	1.7722E+11	2.1013E+03	1.4152E+10	4.1103E+11	1.5236E+08	
3.0E+06	Mejor	2.5217E+10	9.2834E+07	9.3893E+11	5.4200E+02	2.6963E+10	4.1624E+11	1.4317E+08	
	Mediana	4.1791E+10	9.4230E+07	1.1422E+12	1.3284E+03	3.8849E+10	1.0759E+12	2.5117E+08	
	Peor	8.3615E+10	9.5873E+07	1.3657E+12	8.0567E+03	6.4522E+10	2.2238E+12	6.6247E+08	
	Media	4.3535E+10	9.4198E+07	1.1674E+12	2.1429E+03	4.0259E+10	1.1431E+12	2.9521E+08	
	DevSt	1.3038E+10	8.3464E+05	1.3457E+11	2.0556E+03	1.0489E+10	3.9589E+11	1.3359E+08	

Tabla C.9: La media, mediana, desviación estándar, mejor y peor solución obtenida tras realizar 25 ejecuciones de SHADE por función de prueba.

SHADE									
1000D		*f1*	*f2*	*f3*	*f4*	*f5*	*f6*	*f7*	*f8*
1.2E+05	Mejor	3.0051E+08	1.5244E+04	2.1273E+01	3.7072E+10	3.7706E+06	1.0606E+06	1.2916E+08	1.6165E+13
	Mediana	4.2586E+08	1.6299E+04	2.1289E+01	5.5562E+10	5.0728E+06	1.0631E+06	2.1884E+08	5.2042E+13
	Peor	6.5563E+08	1.7514E+04	2.1305E+01	1.0013E+11	5.7210E+06	1.0652E+06	3.6311E+08	2.0346E+14
	Media	4.3294E+08	1.6305E+04	2.1288E+01	5.9577E+10	5.0000E+06	1.0631E+06	2.3728E+08	7.1635E+13
	DevSt	8.1977E+07	6.1020E+02	7.9291E-03	1.6146E+10	4.6944E+05	1.0916E+03	6.7407E+07	4.3396E+13
6.0E+05	Mejor	6.6105E+06	1.4507E+04	2.0610E+01	3.3221E+09	2.3822E+06	1.0558E+06	6.3974E+06	3.2083E+11
	Mediana	1.6050E+07	1.5434E+04	2.0623E+01	5.7704E+09	2.8122E+06	1.0606E+06	9.7469E+06	3.2214E+12
	Peor	4.2061E+07	1.6682E+04	2.0645E+01	9.5292E+09	3.3848E+06	1.0620E+06	1.9021E+07	1.1212E+13
	Media	1.7272E+07	1.5473E+04	2.0624E+01	5.9757E+09	2.8134E+06	1.0601E+06	1.0935E+07	4.1950E+12
	DevSt	8.1680E+06	5.9993E+02	1.0773E-02	1.5604E+09	2.9670E+05	1.4649E+03	3.3854E+06	3.2397E+12
3.0E+06	Mejor	2.1889E+05	1.4414E+04	2.0056E+01	5.5368E+08	1.8960E+06	1.0540E+06	1.5994E+06	4.1572E+10
	Mediana	1.2664E+06	1.5314E+04	2.0060E+01	9.0422E+08	2.3484E+06	1.0560E+06	2.7943E+06	1.0694E+11
	Peor	9.8627E+06	1.6576E+04	2.0065E+01	4.3270E+09	2.7848E+06	1.0586E+06	7.2132E+06	1.0020E+13
	Media	2.1845E+06	1.5379E+04	2.0060E+01	1.1522E+09	2.3572E+06	1.0564E+06	3.1042E+06	8.9442E+11
	DevSt	2.3218E+06	5.8792E+02	2.0426E-03	7.8508E+08	2.4251E+05	1.2587E+03	1.0752E+06	2.4443E+12
1000D		*f9*	*f10*	*f11*	*f12*	*f13*	*f14*	*f15*	
1.2E+05	Mejor	9.6244E+08	9.3318E+07	9.1815E+11	1.1501E+10	3.1816E+09	1.7808E+10	1.6123E+07	
	Mediana	1.9678E+09	9.4261E+07	9.3938E+11	1.6502E+10	6.5857E+09	3.8622E+10	2.1227E+07	
	Peor	3.4032E+09	9.4739E+07	1.0062E+12	2.3104E+10	1.0173E+10	1.0408E+11	3.3401E+07	
	Media	1.9875E+09	9.4152E+07	9.3844E+11	1.6504E+10	6.5152E+09	4.3136E+10	2.2297E+07	
	DevSt	6.4012E+08	3.8878E+05	1.8374E+10	2.9068E+09	1.8308E+09	1.9519E+10	3.9092E+06	
6.0E+05	Mejor	6.2897E+08	9.2664E+07	9.1528E+11	2.9011E+07	3.2133E+08	1.9799E+08	4.0842E+06	
	Mediana	1.4655E+09	9.3501E+07	9.3392E+11	8.2390E+07	6.5380E+08	5.0333E+08	5.0602E+06	
	Peor	2.7380E+09	9.4013E+07	9.9704E+11	7.8442E+08	2.0288E+09	2.8181E+09	1.1867E+07	
	Media	1.5042E+09	9.3474E+07	9.3355E+11	1.2691E+08	7.0836E+08	7.3684E+08	5.5794E+06	
	DevSt	5.3524E+08	3.6050E+05	1.7235E+10	1.4975E+08	3.5285E+08	6.9758E+08	1.6070E+06	
3.0E+06	Mejor	6.1899E+08	9.2053E+07	9.1522E+11	2.5066E+05	7.0364E+06	2.6675E+07	1.1295E+06	
	Mediana	1.3724E+09	9.2736E+07	9.3385E+11	5.1143E+06	2.2551E+07	7.1064E+07	1.3517E+06	
	Peor	2.3575E+09	9.3173E+07	9.9683E+11	5.2344E+07	6.0782E+08	4.5971E+08	2.2615E+06	
	Media	1.3792E+09	9.2711E+07	9.3339E+11	8.2921E+06	5.7585E+07	1.2944E+08	1.4229E+06	
	DevSt	4.6091E+08	3.0474E+05	1.7193E+10	1.1402E+07	1.2439E+08	1.3412E+08	3.0107E+05	

Tabla C.10: La media, mediana, desviación estándar, mejor y peor solución obtenida tras realizar 25 ejecuciones de $eSHADE_{ls}$ por función de prueba.

$eSHADE_{ls}$									
1000D		***f1***	***f2***	***f3***	***f4***	***f5***	***f6***	***f7***	***f8***
	Mejor	3.0608E+06	2.3250E+02	2.0003E+01	1.8822E+11	2.1205E+07	1.0492E+06	2.9570E+09	7.6036E+15
	Mediana	6.3995E+06	2.4934E+02	2.0003E+01	9.2249E+11	2.8597E+07	1.0592E+06	5.5265E+09	5.0167E+16
1.2E+05	Peor	1.3352E+07	2.7374E+02	2.0003E+01	2.3938E+12	4.1072E+07	1.0662E+06	1.7972E+10	1.2457E+17
	Media	6.6170E+06	2.5071E+02	2.0003E+01	9.8731E+11	2.9640E+07	1.0590E+06	7.4300E+09	5.6340E+16
	DevSt	2.0876E+06	1.0383E+01	1.9955E-04	5.4163E+11	5.8575E+06	3.6026E+03	4.1898E+09	2.9608E+16
	Mejor	2.5336E-19	2.1537E-11	2.0000E+01	1.3142E+09	7.6430E+06	1.0484E+06	6.8879E+06	1.1194E+14
	Mediana	6.6604E-06	1.9899E+00	2.0000E+01	6.0848E+09	9.8797E+06	1.0568E+06	6.9152E+07	3.5344E+14
6.0E+05	Peor	8.7119E+01	1.9899E+01	2.0000E+01	1.8805E+10	1.6550E+07	1.0638E+06	5.0124E+08	1.1577E+15
	Media	5.0671E+00	3.3988E+00	2.0000E+01	7.9420E+09	1.0498E+07	1.0562E+06	9.0724E+07	4.1664E+14
	DevSt	1.8356E+01	4.8464E+00	9.1287E-06	5.0862E+09	2.2554E+06	4.1976E+03	1.0010E+08	2.1034E+14
	Mejor	1.2424E-25	0.0000E+00	2.0000E+01	1.0562E+08	5.8927E+06	1.0034E+06	1.1035E+01	1.6857E+13
	Mediana	6.8613E-24	1.9899E+00	2.0000E+01	2.3795E+08	9.7069E+06	1.0409E+06	1.9000E+02	3.9281E+13
3.0E+06	Peor	5.4405E-16	1.9899E+01	2.0000E+01	5.4886E+08	1.6550E+07	1.0596E+06	1.2122E+03	1.1211E+14
	Media	4.0663E-17	3.3829E+00	2.0000E+01	2.5036E+08	1.0247E+07	1.0395E+06	2.6734E+02	4.6405E+13
	DevSt	1.2421E-16	4.8573E+00	0.0000E+00	1.1572E+08	2.4059E+06	1.5544E+04	2.7979E+02	2.6335E+13
1000D		***f9***	***f10***	***f11***	***f12***	***f13***	***f14***	***f15***	
	Mejor	2.5742E+09	9.3110E+07	9.4302E+11	3.2973E+04	2.1708E+10	3.7388E+11	7.5025E+08	
	Mediana	3.7877E+09	9.4404E+07	1.0167E+12	4.4511E+04	3.1863E+10	1.0045E+12	2.7345E+09	
1.2E+05	Peor	7.7658E+09	9.5186E+07	1.6274E+12	5.2838E+04	7.0024E+10	1.9210E+12	1.7381E+10	
	Media	4.2031E+09	9.4347E+07	1.0594E+12	4.4905E+04	3.4874E+10	9.8006E+11	4.3584E+09	
	DevSt	1.3540E+09	5.4426E+05	1.5551E+11	4.8735E+03	1.1632E+10	3.4444E+11	4.2600E+09	
	Mejor	1.5644E+09	9.1744E+07	9.1594E+11	1.3866E+03	6.9438E+08	6.2506E+09	1.3398E+07	
	Mediana	2.4139E+09	9.2926E+07	9.2567E+11	2.4370E+03	2.0943E+09	4.7156E+10	2.6729E+07	
6.0E+05	Peor	4.5665E+09	9.4598E+07	1.0245E+12	8.6651E+03	5.7252E+09	2.4049E+11	6.9937E+07	
	Media	2.4750E+09	9.3047E+07	9.3346E+11	3.2014E+03	2.5832E+09	6.3330E+10	3.0700E+07	
	DevSt	6.3757E+08	7.8272E+05	2.1903E+10	1.9785E+03	1.5341E+09	5.5972E+10	1.7690E+07	
	Mejor	1.2594E+09	9.0760E+07	9.1570E+11	2.0449E+02	5.3631E+03	4.4430E+06	4.2755E+05	
	Mediana	2.2436E+09	9.2068E+07	9.2457E+11	4.0024E+02	1.2117E+05	4.8126E+06	5.6452E+05	
3.0E+06	Peor	4.5665E+09	9.3512E+07	1.0242E+12	5.4262E+02	2.7610E+06	6.0749E+06	8.4453E+05	
	Media	2.3639E+09	9.1982E+07	9.3258E+11	3.6988E+02	2.9525E+05	4.9800E+06	5.8184E+05	
	DevSt	6.2962E+08	8.3130E+05	2.2064E+10	1.1571E+02	5.5739E+05	4.4162E+05	9.3414E+04	

Tabla C.11: La media, mediana, desviación estándar, mejor y peor solución obtenida tras realizar 25 ejecuciones de SHADE-ILS por función de prueba.

SHADE-ILS									
1000D		***f1***	***f2***	***f3***	***f4***	***f5***	***f6***	***f7***	***f8***
	Mejor	1.2300E+04	2.1700E+03	2.0100E+01	1.1200E+10	1.4700E+06	1.0400E+06	2.2300E+08	5.5800E+13
	Mediana	4.4000E+04	2.5700E+03	2.0100E+01	3.8000E+10	2.3100E+06	1.0500E+06	3.5400E+08	1.9600E+14
1.2E+05	Peor	1.0300E+05	2.8100E+03	2.0100E+01	6.9600E+10	3.0500E+06	1.0600E+06	5.7200E+08	5.4900E+14
	Media	5.1296E+04	2.5452E+03	2.0100E+01	3.5864E+10	2.3556E+06	1.0516E+06	3.6856E+08	2.3631E+14
	DevSt	2.7921E+04	1.4142E+02	0.0000E+00	1.5187E+10	3.6658E+05	6.2450E+03	9.7432E+07	1.4782E+14
	Mejor	0.0000E+00	1.4300E+03	2.0000E+01	1.9500E+09	1.4000E+06	1.0300E+06	7.8600E+05	1.9400E+12
	Mediana	5.8800E-24	1.7400E+03	2.0100E+01	3.6900E+09	2.0900E+06	1.0400E+06	1.4200E+06	1.4400E+13
6.0E+05	Peor	2.1200E-22	2.1400E+03	2.0100E+01	5.8300E+09	2.9500E+06	1.0500E+06	2.9200E+06	3.1500E+13
	Media	3.5512E-23	1.7864E+03	2.0088E+01	3.7448E+09	2.1032E+06	1.0424E+06	1.5421E+06	1.4339E+13
	DevSt	6.3958E-23	1.9213E+02	3.3166E-02	1.0749E+09	3.8091E+05	7.2342E+03	5.8583E+05	7.1935E+12
	Mejor	0.0000E+00	9.1500E+02	2.0000E+01	1.4800E+08	8.0700E+05	1.0100E+06	1.9800E+01	8.6500E+10
	Mediana	0.0000E+00	1.0300E+03	2.0100E+01	2.8900E+08	1.3800E+06	1.0300E+06	1.2400E+02	4.5900E+11
3.0E+06	Peor	1.9200E-27	1.3000E+03	2.0100E+01	5.2800E+08	1.6800E+06	1.0500E+06	9.8700E+02	2.0100E+12
	Media	2.5558E-28	1.0415E+03	2.0068E+01	3.0128E+08	1.3310E+06	1.0316E+06	2.2356E+02	5.9937E+11
	DevSt	5.3619E-28	1.0341E+02	4.7610E-02	1.0458E+08	2.2657E+05	9.8658E+03	2.5286E+02	5.4287E+11
1000D		***f9***	***f10***	***f11***	***f12***	***f13***	***f14***	***f15***	
	Mejor	2.0800E+08	9.2200E+07	1.8300E+09	2.2500E+03	3.7900E+09	6.9800E+10	5.2200E+07	
	Mediana	2.7900E+08	9.4200E+07	3.9500E+09	2.5400E+03	1.3700E+10	1.4700E+11	8.9500E+07	
1.2E+05	Peor	3.5400E+08	9.4800E+07	1.6200E+10	3.3600E+03	2.8400E+10	3.9100E+11	1.3600E+08	
	Media	2.8760E+08	9.3956E+07	5.5884E+09	2.6464E+03	1.3623E+10	1.8416E+11	8.8616E+07	
	DevSt	3.7500E+07	6.4036E+05	3.8645E+09	3.0895E+02	6.7078E+09	9.0309E+10	2.4589E+07	
	Mejor	1.9200E+08	9.1800E+07	6.5100E+07	1.0800E+03	7.1400E+07	6.1600E+07	4.8300E+06	
	Mediana	2.4600E+08	9.3500E+07	1.2300E+08	1.7300E+03	6.2200E+08	2.3200E+08	1.1000E+07	
6.0E+05	Peor	3.3400E+08	9.4200E+07	2.1400E+08	2.3000E+03	1.0700E+09	2.0100E+09	4.7000E+07	
	Media	2.4932E+08	9.3192E+07	1.3048E+08	1.7688E+03	5.6046E+08	4.9090E+08	1.6893E+07	
	DevSt	3.7174E+07	6.3831E+05	3.4470E+07	2.7114E+02	3.1299E+08	6.2008E+08	1.3683E+07	
	Mejor	1.3200E+08	9.1800E+07	2.4700E+05	7.7900E-20	1.0800E+05	6.1300E+06	3.7900E+05	
	Mediana	1.5500E+08	9.2600E+07	5.1000E+05	1.5900E-01	1.0300E+06	7.1900E+06	6.4500E+05	
3.0E+06	Peor	1.8600E+08	9.3500E+07	1.1100E+06	1.0400E+03	2.8400E+06	1.0900E+07	3.0700E+06	
	Media	1.5780E+08	9.2556E+07	5.3888E+05	6.4886E+01	1.0706E+06	7.6280E+06	8.6832E+05	
	DevSt	1.4888E+07	4.5100E+05	2.2303E+05	2.2987E+02	8.6269E+05	1.2756E+06	6.3444E+05	

D | Prueba de suma de rangos de Wilcoxon

En este apéndice se muestran los resultados obtenidos al aplicar la **prueba de suma de rangos de Wilcoxon** entre la nueva propuesta y cada uno de sus componentes además del algoritmo de referencia SHADE-ILS. Las pruebas son realizada con $\boldsymbol{N = 25}$ y $\boldsymbol{p = 0.05}$, lo que significa que se utiliza un tamaño de muestra de 25 ejecuciones y un nivel de significancia estadística del 5 %. Los datos mostrados en cada tabla corresponden a los **valores-$\boldsymbol{p}$** calculados mediante la función `stats.wilcoxon()` del lenguaje de programación `Python` donde `stats` es un paquete importado de la biblioteca `scipy`. Un valor mayor a 5.00E-02 indica que no existe suficiente evidencia como para afirmar que existe una diferencia significativa entre el desempeño promedio de GL-SHADE y el desempeño promedio del algoritmo contra el cual se le está comparando.

Tabla D.1: Prueba de suma de rangos de Wilcoxon con $N = 25$ y $p = 0.05$: GL-SHADE vs. {MTS-LS1, SHADE, eSHADE$_{\text{ls}}$, SHADE-ILS} empleando el conjunto de funciones de prueba @1.2E+05 evaluaciones de la función objetivo.

GL-SHADE es el algoritmo de control				
Función	**MTS-LS1**	**SHADE**	**eSHADE$_{\text{ls}}$**	**SHADE-ILS**
f_1	1.23E-05	1.23E-05	1.23E-05	1.23E-05
f_2	1.23E-05	1.23E-05	1.23E-05	1.23E-05
f_3	6.64E-04	1.23E-05	7.79E-02	1.23E-05
f_4	1.23E-05	6.77E-01	1.23E-05	1.57E-03
f_5	1.23E-05	3.28E-04	1.23E-05	1.23E-05
f_6	6.31E-03	1.23E-05	4.46E-04	7.80E-02
f_7	1.23E-05	1.23E-05	1.23E-05	1.23E-05
f_8	1.23E-05	1.23E-05	1.23E-05	8.71E-03
f_9	1.23E-05	4.93E-03	4.57E-05	1.23E-05
f_{10}	5.44E-02	1.86E-02	5.82E-03	3.26E-01
f_{11}	1.23E-05	1.35E-01	5.76E-05	1.23E-05
f_{12}	1.23E-05	1.23E-05	1.23E-05	1.23E-05
f_{13}	1.39E-05	1.23E-05	2.26E-03	7.22E-05
f_{14}	1.23E-05	1.23E-05	1.23E-05	3.28E-04
f_{15}	1.23E-05	1.23E-05	1.23E-05	2.96E-04

Tabla D.2: Prueba de suma de rangos de Wilcoxon con $N = 25$ y $p = 0.05$: GL-SHADE vs. {MTS-LS1, SHADE, eSHADE$_{ls}$, SHADE-ILS} empleando el conjunto de funciones de prueba @6.0E+05 evaluaciones de la función objetivo.

GL-SHADE es el algoritmo de control				
Function	**MTS-LS1**	**SHADE**	**eSHADE$_{ls}$**	**SHADE-ILS**
f_1	1.23E-05	1.23E-05	1.57E-04	1.23E-05
f_2	1.23E-05	1.23E-05	1.23E-05	1.23E-05
f_3	1.77E-05	1.23E-05	1.21E-05	2.44E-05
f_4	1.23E-05	7.33E-04	5.45E-04	1.43E-01
f_5	1.23E-05	1.22E-01	1.23E-05	1.26E-04
f_6	1.38E-02	2.54E-05	3.70E-02	7.22E-05
f_7	1.23E-05	1.57E-05	2.96E-04	1.23E-05
f_8	1.23E-05	3.28E-04	1.23E-05	1.35E-01
f_9	1.23E-05	1.57E-04	7.57E-01	1.23E-05
f_{10}	2.26E-05	1.57E-04	1.83E-01	8.04E-03
f_{11}	1.23E-05	2.64E-01	1.35E-01	1.23E-05
f_{12}	5.45E-04	1.23E-05	2.26E-05	1.23E-05
f_{13}	1.23E-05	1.23E-05	4.43E-01	1.23E-05
f_{14}	1.23E-05	1.39E-05	1.77E-05	1.39E-05
f_{15}	1.23E-05	1.23E-05	7.98E-01	2.70E-03

Tabla D.3: Prueba de suma de rangos de Wilcoxon con $N = 25$ y $p = 0.05$: GL-SHADE vs. {MTS-LS1, SHADE, eSHADE$_{ls}$, SHADE-ILS} empleando el conjunto de funciones de prueba @3.0E+06 evaluaciones de la función objetivo.

GL-SHADE es el algoritmo de control				
Function	**MTS-LS1**	**SHADE**	**eSHADE$_{ls}$**	**SHADE-ILS**
f_1	2.31E-01	1.23E-05	2.64E-02	1.23E-05
f_2	1.23E-05	1.23E-05	1.28E-01	1.23E-05
f_3	1.23E-05	1.23E-05	1.00E+00	3.74E-05
f_4	1.23E-05	1.23E-05	1.23E-05	1.23E-05
f_5	1.23E-05	2.31E-01	1.23E-05	1.23E-05
f_6	2.54E-05	1.23E-05	1.74E-01	5.27E-01
f_7	1.23E-05	1.23E-05	1.23E-05	1.23E-05
f_8	1.23E-05	4.93E-03	1.23E-05	5.76E-05
f_9	1.23E-05	2.26E-05	9.04E-01	1.23E-05
f_{10}	1.23E-05	2.26E-05	1.66E-01	2.26E-05
f_{11}	1.23E-05	3.13E-01	3.82E-01	1.23E-05
f_{12}	1.23E-05	1.23E-05	1.23E-05	1.87E-04
f_{13}	1.23E-05	1.23E-05	8.09E-05	1.23E-05
f_{14}	1.23E-05	1.23E-05	5.11E-02	1.23E-05
f_{15}	1.23E-05	4.27E-01	2.30E-02	1.66E-01

Bibliografía

[1] Swagatam Das and Ponnuthurai Nagaratnam Suganthan. Differential evolution: A survey of the state-of-the-art. *IEEE Transactions on Evolutionary Computation*, 15(1):4–31, February 2011.

[2] Alberto Cano and Carlos García-Martínez. 100 Million dimensions large-scale global optimization using distributed GPU computing. In *2016 IEEE Congress on Evolutionary Computation (CEC'2016)*, pages 3566–3573, Vancouver, British Columbia, Canada, 24-29 July 2016. IEEE Press. ISBN 978-1-5090-0624-3.

[3] Antonio LaTorre, Santiago Muelas, and José-María Pe na. Large scale global optimization: Experimental results with MOS-based hybrid algorithms. In *2013 IEEE Congress on Evolutionary Computation*, pages 2742–2749, Cancún, México, 20-23 June 2013. IEEE Press. ISBN 978-1-4799-0453-2.

[4] (Jun-Rong Jian, Zhi-Hui Zhan, and Jun Zhang. Large-scale evolutionary optimization: a survey and experimental comparative study. *International Journal of Machine Learning and Cybernetics*, 11(3):729–745, March 2020.

[5] Carlos A. Coello Coello, Gary B. Lamont, and David A. Van Veldhuizen. *Evolutionary Algorithms for Solving Multi-Objective Problems*. Springer, New York, second edition, September 2007. ISBN 978-0-387-33254-3.

[6] Daniel Molina, Antonio LaTorre, and Francisco Herrera. SHADE with Iterative Local Search for Large-Scale Global Optimization. In *2018 IEEE Congress on Evolutionary Computation*, Rio de Janeiro, Brazil, 8-13 July 2018. IEEE Press. ISBN 978-1-5090-6018-4.

[7] Anas A. Hadi, Ali W. Mohamed, and Kamal M. Jambi. LSHADE-SPA memetic framework for solving large-scale optimization problems. *Complex & Intelligent Systems*, 5(1):25–40, March 2019.

[8] Daniel Molina and Francisco Herrera. Iterative hybridization of DE with local search for the CEC'2015 special session on large scale global optimization. In *2015 IEEE*

Congress on Evolutionary Computation (CEC'2015), pages 1974–1978, Sendai, Japan, 25-28 May 2015. IEEE Press. ISBN 978-1-4799-7492-4.

[9] Zhenyu Yang, Ke Tang, and Xin Yao. Large scale evolutionary optimization using cooperative coevolution. *Information Sciences*, 178:2985–2999, August 1 2008.

[10] Tetsuyuki Takahama and Setsuko Sakai. Large scale optimization by differential evolution with landscape modality detection and a diversity archive. In *2012 IEEE Congress on Evolutionary Computation (CEC'2012)*, Brisbane, Australia, 10-15 June 2012. ISBN 978-1-4673-1510-4.

[11] Ali Wagdy Mohamed. Solving large-scale global optimization problems using enhanced adaptive differential evolution algorithm. *Complex & Intelligent Systems*, 3:205–231, 2017.

[12] Xiangjuan Wu, Yuping Wang, Junhua Liu, and Ningle Fan. A New Hybrid Algorithm for Solving Large Scale Global Optimization Problems. *IEEE Access*, 7:103354–103364, 2019.

[13] Mohammad Nabi Omidvar, Xiaodong Li, Yi Mei, and Xin Yao. Cooperative Co-Evolution With Differential Grouping for Large Scale Optimization. *IEEE Transactions on Evolutionary Computation*, 18(3):378–393, June 2014.

[14] Thomas Weise. *Global Optimization Algorithms —Theory and Application—*.

[15] Panos M. Pardalos and H. Edwin Romeijn, editors. *Handbook of Global Optimization. Volume 2*. 2002. ISBN 978-1402006326.

[16] Jos'e Marcos Moreno Vega, María Belén Melián Batista, and José Andrés Moreno Pérez. Metaheurísticas: Una visión global. *Inteligencia artificial: Revista Iberoamericana de Inteligencia Artificial*, 7(19):7–28, 2003.

[17] Amín Vanya Bernabé Rodríguez. Diseño de una nueva función de escalarización usando programación genética. Master's thesis, CINVESTAV-IPN, Departamento de Computación, Ciudad de México, México, Diciembre 2019.

[18] Hanan Hiba, Mohammed El-Abd, and Shahryar Rahnamayan. Improving SHADE with Center-based Mutation for Large-scale Optimization. In *2019 IEEE Congress on Evolutionary Computation (CEC'2019)*, pages 1533–1540, Wellington, New Zealand, 10-13 June 2019. IEEE Press. ISBN 978-1-7281-2154-3.

[19] Lin-Yu Tseng and Chun Chen. Multiple trajectory search for Large Scale Global Optimization. In *2008 IEEE Congress on Evolutionary Computation (CEC'2008)*, pages 3052–3059, Hong Kong, 1-6 June 2008. IEEE Press. ISBN 978-1-4244-1822-0.

[20] Mitchell A. Potter and Kenneth A. De Jong. A Cooperative Coevolutionary Approach to Function Optimization. In Yuval Davidor, Hans-Paul Schwefel, and Reinhard Männer, editors, *Parallel Problem Solving from Nature — PPSN III, International Conference on Evolutionary Computation*, pages 249–257, Jerusalem, Israel, October 9-14 1994. Springer. Lecture Notes in Computer Science Vol. 866. ISBN 3-540-58484-6.

[21] Mohammmad N. Omidvar and Xiaodong Li. Cooperative Coevolutionary Algorithms for Large Scale Optimisation. Available at: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.177.560&rep=rep1&type=pdf`, 2010.

[22] Xiangjuan Wu, Yuping Wang, Junhua Liu, and Ninglei Fan. A New Hybrid Algorithm for Solving Large Scale Global Optimization Problems. *IEEE Access*, 7:103354–103364, 29 July 2019.

[23] Yuan Sun, Xiaodong Li, Andreas Ernst, and Mohammad Nabi Omidvar. Decomposition for Large-scale Optimization Problems with Overlapping Components. In *2019 IEEE Congress on Evolutionary Computation (CEC'2019)*, pages 326–333, Wellington, New Zealand, 10-13 June 2019. IEEE Press. ISBN 978-1-7281-2154-3.

[24] Morteza Husainy Yar, Vahid Rahmati, and Hamid Reza Dalili Oskouei. A Survey on Evolutionary Computation: Methods and Their Applications in Engineering. *Modern Applied Science*, 10(11):131–139, 2016.

[25] D.B. Fogel. An introduction to simulated evolutionary optimization. *IEEE Transactions on Neural Networks*, 5(1):3–14, January 1994.

[26] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, USA, 1997. ISBN 978-0-07-042807-2.

[27] Hans-Georg Beyer and Hans-Paul Schwefel. Evolution strategies—a comprehensive introduction. *Natural Computing*, 1:3–52, 2002.

[28] Rainer Storn and Kenneth Price. Differential Evolution – A Simple and Efficient Heuristic for global Optimization over Continuous Spaces. *Journal of Global Optimization*, 11(4):341–359, December 1997.

[29] Wang li Xiang, Xue lei Meng, Mei qing An, Yinzhen Li, and Ming xia Gao. An Enhanced Differential Evolution Algorithm Based on Multiple Mutation Strategies. *Computational Intelligence and Neuroscience*, 2015, 2015. Article number: 285730.

[30] A.K. Qin, V.L. Huang, and P.N. Suganthan. Differential Evolution Algorithm With Strategy Adaptation for Global Numerical Optimization. *IEEE Transactions on Evolutionary Computation*, 13(2):398–417, April 2009.

[31] Yong Wang, Zixing Cai, and Qingfu Zhang. Differential Evolution with Composite Trial Vector Generation Strategies and Control Parameters. *IEEE Transactions on Evolutionary Computation*, 15(1):55–66, February 2011.

[32] Rammohan Mallipeddi and Ponnuthurai Nagaratnam Suganthan. Differential Evolution Algorithm with Ensemble of Parameters and Mutation and Crossover Strategies. In Bijaya Ketan Panigrahi, Swagatam Das, Ponnuthurai Nagaratnam Suganthan, and Subhransu Sekhar Dash, editors, *Swarm, Evolutionary, and Memetic Computing. First International Conference on Swarm, Evolutionary, and Memetic Computing, SEMCCO 2010*. Springer. Lecture Notes in Computer Science Vol. 6466, Chennai, India, December 16-18 2010. ISBN 978-3-642-17562-6.

[33] Ryoji Tanabe and Alex Fukunaga. Success-history based parameter adaptation for Differential Evolution. In *2013 IEEE Congress on Evolutionary Computation (CEC'2013)*, pages 71–78, Cancún, México, 20-23 June 2013. IEEE Press. ISBN 978-1-4799-0453-2.

[34] Jingqiao Zhang and Arthur C. Sanderson. JADE: Adaptive Differential Evolution With Optional External Archive. *IEEE Transactions on Evolutionary Computation*, 13(5):945–.958, October 2009.

[35] Janez Brest, Saso Greiner, Borko Boskovic, Marjan Mernik, and Vijern Zumer. Self-Adapting Control Parameters in Differential Evolution: A Comparative Study on Numerical Benchmark Problems. *IEEE Transactions on Evolutionary Computation*, 10(6):646–657, December 2006.

[36] Zhenyu Yang, Ke Tang, and Xin Yao. Multilevel cooperative coevolution for large scale optimization. In *2008 IEEE Congress on Evolutionary Computation (CEC'2008)*, pages 1663–1670, Hong Kong, 1-6 June 2008. IEEE Press. ISBN 978-1-4244-1822-0.

[37] Xiaodong Li, Ke Tang, Mohammad N. Omidvar, Zhenyu Yang, and Kai Qin. Benchmark Functions for the CEC'2013 Special Session and Competition on Large-Scale Global Optimization. Technical report, Evolutionary Computation and Machine Learning Group, RMIT University, Australia, December 24 2013.

[38] Daniel Molina and Antonio LaTorre. WCCI 2018 Large-Scale Global Optimization Competition Results, 2018. `http://www.tflsgo.org/download/comp2018_slides.pdf`, Last accessed on 2020-02-29.

[39] Daniel Molina and Antonio LaTorre. CEC 2019 Large-Scale Global Optimization Competition Results, 2019. `http://www.tflsgo.org/assets/cec2019/comp2019_slides.pdf`, Last accessed on 2020-02-29.

[40] Richard H. Byrd, Peihuang Lu, Jorge Nocedal, and Ciyou Zhu. A Limited Memory Algorithm for Bound Constrained Optimization. *SIAM Journal on Scientific Computing*, 16(5):1190–1208, September 1995.

[41] Ke Tang, Xiaodong Li, P. N. Suganthan, Zhenyu Yang, and Thomas Weise. Benchmark Functions for the CEC'2010 Special Session and Competition on Large-Scale Global Optimization. Technical report, Nature Inspired Computation and Applications Laboratory, University of Science and Technology, China, January 2010.

[42] Borhan Kazimipour, Xiaodong Li, and A.K. Qin. Effects of population initialization on differential evolution for large scale optimization. In *2014 IEEE Congress on Evolutionary Computation (CEC'2014)*, pages 2404–2411, Beijing, China, 6-11 July 2014. ISBN 978-1-4799-6626-4.

[43] K. Tang, X. Yao, P.N. Suganthan, C. MacNish, Y.P. Chen, C.M. Chen, and Z. Yang. Benchmark Functions for the CEC'2008 Special Session and Competition on Large Scale Global Optimization. Technical report, Nature Inspired Computation and Applications Laboratory, University of Science and Technology, China, November 2007.

[44] Daniel Molina and Antonio LaTorre. Special Session and Competition on Large-Scale Global Optimization, 2019. `http://www.tflsgo.org/special_sessions/cec2019`, Last accessed on 2020-05-22.

[45] Daniel Molina and Antonio LaTorre. Toolkit for the Automatic Comparison of Optimizers (TACO): Herramienta *online* avanzada para comparar metaheurísticas. In *XIII Congreso Español en Metaheurísticas y Algoritmos Evolutivos Bioinspirados*, pages 727–732. Asociación Española para la Inteligencia Artificial, 23-26 October 2018. ISBN 978-84-09-05643-9.

[46] Ramiro Marco Figuera. Analysis of illumination conditions at the lunar south pole using parallel computing techniques. Master's thesis, Technische Universität Berlin. Department of Geodesy and Geoinformation Science, Germany, November 2014.

[47] Pavel Krömer, Jan Platoš, Václav Snášel, and Ajith Abraham. Many-Threaded Differential Evolution on the GPU. In Shigeyoshi Tsutsui and Pierre Collet, editors, *Massively Parallel Evolutionary Computation on GPGPUs*, pages 121–147. Springer, Berlin, Germany, 2013. ISBN 978-3-642-37958-1.

[48] NVIDIA. CUDA documentation, 2018. https://docs.nvidia.com/cuda/, Last accessed on 2020-06-29.

[49] Lucas de P. Veronese and Renato A. Krohling. Differential evolution algorithm on the GPU with C-CUDA. In *2010 IEEE Congress on Evolutionary Computation*

(CEC'2010), Barcelona, Spain, 18-23 July 2010. IEEE Press. ISBN 978-1-4244-6909-3.

[50] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, Upper Saddle, New Jersey, USA, 2010. ISBN 978-0131387683.